GUARD GENERATION FOR A DISTRIBUTED WORKFLOW ENACTMENT SERVICE

A THESIS SUBMITTED TO
THE GRADUATE SCHOOL OF NATURAL AND APPLIED SCIENCES
OF
THE MIDDLE EAST TECHNICAL UNIVERSITY

BY

NESİME TATBUL

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE

DEGREE OF

MASTER OF SCIENCE

IN

THE DEPARTMENT OF COMPUTER ENGINEERING

JANUARY 1998

Approval of the Graduate School of Natural and Applied Sciences.

Prof. Dr. Tayfur Öztürk
Director

I certify that this thesis satisfies all the requirements as a thesis for the degree of Master of Science.

Prof. Dr. Fatoş Yarman Vural
Head of Department

This is to certify that we have read this thesis and that in our opinion it is fully adequate, in scope and quality, as a thesis for the degree of Master of Science.

Prof. Dr. Asuman Doğaç
Supervisor

Examining Committee Members

Prof. Dr. Asuman Doğaç _____

Assoc. Prof. Dr. Aral Ege _____

Assoc. Prof. Dr. Özgür Ulusoy _____

Assoc. Prof. Dr. İsmail Hakkı Toroslu _____

Assist. Prof. Dr. Ahmet Coşar _____

ABSTRACT

GUARD GENERATION FOR A DISTRIBUTED WORKFLOW ENACTMENT SERVICE

Tatbul, Nesime

M.S., Department of Computer Engineering

Supervisor: Prof. Dr. Asuman Doğaç

January 1998, 82 pages

Workflows are activities involving the coordinated execution of multiple tasks performed by different processing entities. Since they execute mostly in distributed heterogeneous environments involving a variety of human and system tasks, distributed scheduling of workflows is essential. A distributed workflow enactment service contains several schedulers on different nodes of a network each executing parts of process instances. Such an architecture fits naturally to the distributed heterogeneous environments. Furthermore, distributed enactment service provides failure resiliency and increased performance since a centralized scheduler is a potential bottleneck.

In this thesis, a guard-based distributed workflow enactment service is introduced in which distributed scheduling of activities is achieved through guard expressions. Guards are temporal expressions defined on events such that events can happen only if their guards are true. This thesis presents how intertask dependencies, i.e. the control flow between the tasks, can be incorporated into simple temporal expressions and the guard generation algorithm that constructs all the guards of a workflow process from the workflow specification.

Keywords: workflow, workflow management system, workflow definition language, process, process tree, block, activity, task, intertask dependency, guard, guard generation, guard handler, CORBA, distributed enactment service, ACTA Formalism

ÖZ

DAĞITIK BİR İŞ AKIŞI HAREKETE GEÇİRME SERVİSİ İÇİN NÖBETÇİ OLUŞTURULMASI

Tatbul, Nesime

Yüksek Lisans, Bilgisayar Mühendisliği Bölümü

Tez Yöneticisi: Prof. Dr. Asuman Doğaç

Ocak 1998, 82 sayfa

İş akışları farklı işlem birimleri tarafından yapılan, çok kısımdan oluşan işlerin koordineli çalışmasını içeren faaliyetlerdir. Çoğunlukla insan ve sistem tarafından yapılan işlerden oluşan dağıtık ve heterojen çevrelerde çalıştıklarından, iş akışlarının dağıtık planlanması gereklidir. Dağıtık bir iş akışı harekete geçirme servisi, şebekenin farklı bölümlerinde iş akışı örneğinin parçalarını çalıştıran farklı planlayıcılar içerir. Böyle bir yapı, dağıtık ve heterojen ortamlara doğal olarak uygundur. Ayrıca, dağıtık bir iş akışı harekete geçirme servisi hataya karşı esneklik ve arttırılmış performans sağlar. Çünkü, merkezi planlayıcı potansiyel bir darboğaz teşkil eder.

Bu tezde, faaliyetlerin dağıtık planlanmasının nöbetçi ifadeler tarafından yapıldığı dağıtık bir iş akışı harekete geçirme servisi öne sürülmektedir. Nöbetçiler olaylar üzerine tanımlanmış ifadelerdir. Olaylar sadece nöbetçi ifadeleri doğru ise gerçekleşebilirler. Bu tez, işler arasındaki bağımlılıkların (işler arasındaki kontrol akışının) nasıl basit zaman belirten nöbetçi ifadelere dönüştürüldüğünü ve iş akışı tanımından yola çıkılarak bir iş akışı işleminin bütün nöbetçi ifadelerini oluşturan

bir algoritmayı sunmaktadır.

Anahtar Kelimeler: iş akışı, iş akışı yönetim sistemi, iş akışı tanımlama dili, işlem, işlem ağacı, blok, faaliyet, iş, işlerarası bağımlılık, nöbetçi, nöbetçi oluşturulması, nöbetçi yöneticisi, CORBA, dağıtık harekete geçirme servisi, ACTA Formalizmi

To My Parents

ACKNOWLEDGMENTS

I am grateful to many individuals for the cooperation, support and the encouragement they gave me while preparing this thesis. At top of all, I am honored to present my special thanks to my supervisor Prof. Dr. Asuman Doğaç without whose continuous guidance and encouragement this work could have never been possible.

I would like to thank to my partner Esin Gökkoca for her sincere and unforgettable cooperation and help. I would also like to thank to other members of METUFlow project for their friendship and support; Pınar Köksal, Pınar Karagöz, Sena Nural Arpınar and Budak Arpınar.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

A workflow can be defined as a collection of processing steps (also termed as tasks or activities) organized to accomplish some business process. A task can be performed by one or more software systems, or, by a person or a team, or a combination of these. In addition to the collection of tasks, a workflow defines the order of task invocation or condition(s) under which tasks must be invoked, i.e. control flow, and data flow between these tasks.

Workflow management is the automated coordination, control and communication of work as is required to satisfy workflow processes. Workflow Management System (WFMS) is a system that completely defines, manages and executes workflows through the execution of software whose order of execution is driven by a computer representation of the workflow logic [19]. Workflow management systems aim at automating business processes to provide flexibility to cope with on going business changes. Furthermore, WFMSs coordinate and streamline complex business processes within large organizations to achieve improvements in critical, contemporary measures of performances, such as cost, quality, service and speed.

There are a number of commercial workflow products in the market today. In spite of their initial success, these systems are far from meeting the demands of today's complex organizations. Current workflow systems are complex to install, use and maintain, have only limited resilience to failures, have poor scalability

and are inflexible to cope with the multitude of different application environments where they are being deployed. Furthermore, existing WFMSs fail to satisfy the user expectations in several respects including the following: handling of heterogeneity and interoperability, providing support for advanced transaction models, run-time flexibility, role-oriented flexible worklist management and history tracking, supporting mobile users and advanced security mechanisms.

Workflow systems mostly execute in distributed, autonomous and heterogeneous environments involving a variety of human and system tasks which are very common in enterprises of even moderate complexity. To provide the coordinated execution of tasks in a workflow running in such an environment, a distributed scheduling mechanism is essential which will handle the instantiation of tasks according to the conditions that determine the control flow between the tasks. Control flow in workflows is determined by the dependencies between tasks. The core component of a WFMS is the workflow enactment service which is responsible for the scheduling of tasks. Workflow enactment service instantiates processes according to the process description and controls correct execution of activities interacting with users and other components of the system as necessary.

Distributed scheduling of workflows has been addressed in [1, 3, 37]. In [1], a distributed workflow system is proposed based on persistent message queues where the authors assume that the processes are well-formed, i.e., they do not have cycles or dependencies that may compromise their execution. The authors further assume that the data flow follows the ordering imposed by the control flow to avoid race conditions. [3] proposes INCAs for distributed workflow management. In this model, each execution of a process is associated with an INformation CArrier (INCA), which is an object that contains all the necessary information for the execution as well as propagation of the object among the relevant processing nodes. [37] brings a more feasible solution which we used as the basis of our distributed scheduling mechanism. According to [37], the main activity of a workflow is organization and coordination of tasks which might be dependent on each other by their states. Since state changes of the tasks are represented as event generations, state dependencies can be defined through event ordering. Therefore,

controlling the occurrences of events provides the coordination of the tasks. To make distributed execution of workflow computations possible, occurrences of events are not controlled in a central scheduler. Instead, each event is made responsible for controlling its execution to decide on the right time to occur. Guards, which are logical expressions defined on events, are used for this purpose. Occurrences of events are permitted only if their guards are true.

To achieve such a coordinated execution between the activities, the intertask dependencies should be clearly introduced into the workflow system. We designed a block-structured workflow specification language in which block constructs are used to express intertask dependencies. After a workflow is specified using this language, it is parsed into guard expressions to reveal the dependency information in the block constructs. Then we create a guard handler for each of the activities in the workflow. A guard handler is a functional unit of the scheduler that contains the guard expressions for the significant events (like start, commit or abort) of the activity the guard handler belongs to [17]. The evaluation and the management of guard expressions are accomplished by the guard handler so that scheduling of the activities in the system is provided.

In this thesis, we describe the guard generation for a distributed workflow enactment service in which the distributed scheduling of the activities are achieved through the use of guard expressions. Our enactment service is based on the work presented in [37]. In order to confine the theory presented in [37] to a manageable practical implementation, we started by designing a block-structured procedural workflow specification language. In this way it becomes possible to express the workflow specification with a well-defined set of dependencies. We show that these dependencies produce simple guard expressions which in turn makes it possible to give a simple algorithm to generate the guards from the specification language.

The block structured nature of the specification language makes it also possible to locate and handle the deadlocks and race conditions without the need for preprocessing the specification. Furthermore, in our workflow specification language, because of its well-defined semantics, the references to the future are known at compile time and can thus be easily handled by a special software

module (a modified 2PC protocol implementation).

The thesis is organized as follows: Chapter 2 summarizes the related work about the workflow management systems. The architecture of METUFlow is provided in Chapter 3. METUFlow is a distributed workflow management system prototype implemented in the light of our research. In Chapter 4, how intertask dependencies are incorporated into guards, how these guards are generated from the workflow specification and how they are further used during execution are explained. Chapter 5 explains the implementation of the guard generation algorithm describing the tools and data structures used. Finally, Chapter 6 concludes the thesis discussing the future work. The Backus-Naur Form of the workflow definition language is also provided in the Appendices.

CHAPTER 2

RELATED WORK

Our primary aim in designing the METUFlow Workflow Management System was to handle the problems of the current workflow management systems like the lack of mechanisms to meet the need for distributed scheduling and distributed execution, heterogeneity and interoperability. In this chapter, first the technologies and standards that we exploited to realize our goal are summarized in Section 2.1, namely CORBA (Common Object Request Broker Architecture) and the Workflow Management Coalition's Reference Model. Some existing workflow models are briefly reviewed in Section 2.2. Finally, in Section 2.3, previous work on workflow scheduling which includes an approach that we have taken as reference is discussed.

2.1 Enabling Technologies and Standards

Workflows are composite activities that typically involve a variety of computational and human tasks and span multiple systems. Workflows arise naturally in heterogenous environments consisting of a variety of databases and information systems. In a heterogeneous environment, applications autonomously developed at different sites in different languages on different hardware and software platforms need to share information and invoke each other's services. It is common that workflow systems access heterogeneous resources and interoperate with other

workflow systems. If a workflow system tries to overcome heterogeneity problem, which can take the form of communication level, platform level or semantic level heterogeneity, within its own architecture, the system becomes very complex and inflexible. Therefore, it is more efficient to base a WFMS framework on a standard middleware that hides some levels of heterogeneity in the environment. The object technology and the distributed computing technology are the enabling technologies to provide necessary communication infrastructure for this purpose. Two well-known distributed object technologies are Object Management Group's CORBA and Microsoft's DCOM. The architecture of METUFlow system is based on OMG's CORBA. This architecture and the facilities that it can provide to create an interoperable workflow system are described in section 2.1.1.

In addition to the CORBA technology, a partial solution to semantic interoperability problem specific to WFMSs can be obtained by complying to the standards being developed by Workflow Management Coalition (WfMC). WfMC, which aims at standardizing the terminology and interoperability between workflow products, defines a Workflow Reference Model which is summarized in Section 2.1.2.

2.1.1 OMG and CORBA Technology

Earlier client/server architectures such as RPC do not have an object-oriented model. Also a client needs to know the location of the server and how to access the services. A client code must be changed whenever it needs to use new services. Convergence of object-oriented programming paradigm and distributed computing technology has resulted in distributed object systems [28].

Object Management Group (OMG) is a consortium of object technology vendors, specifying the architecture for an interoperable system in which the components communicate with each other through a location transparent common middleware. OMG has defined Object Management Architecture (OMA), a standard architecture that joins distributed computing and object-oriented programming technologies [38]. The OMA object model supports encapsulation, abstraction and polymorphism through its object-oriented approach. OMA has four basic

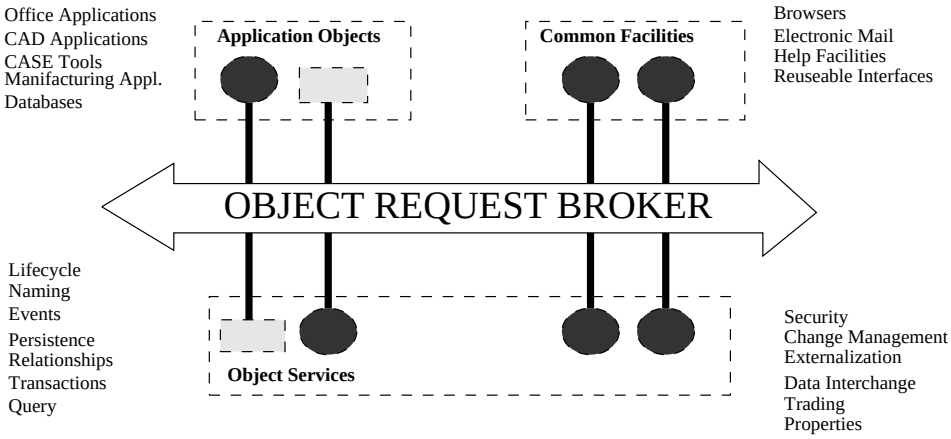


Figure 2.1: OMA Model

components. (1) Object Request Broker is the object interconnection middleware in which clients are insulated from the mechanisms used to communicate with server objects. (2) Common Object Services (COSS) are components with IDL-specified interfaces which extend the capabilities of the ORB. They are complementary standards for integrating distributed objects. (3) Common Facilities are optional IDL-defined components that are useful in many application domains and which are made available through OMA-compliant object interfaces. (4) Application Objects are specific to end-user applications. To participate in ORB-mediated exchanges, they must be defined using IDL (Interface Definition Language).

Object Services, Common Facilities and Application Objects all intercommunicate using the Object Request Broker. The OMA reference model is shown in Figure 2.1.

Object Request Broker (ORB) is the middleware bus that lets clients invoke methods on remote objects either statically or dynamically. Clients send requests to the ORB asking for certain services to be performed by whichever server can fulfill those tasks. The structure of CORBA is shown in Figure 2.2. In CORBA, clients and servers only know the interfaces of the components. The only means of communication is the requests from clients and their responses from servers. In this way a distributed, heterogeneous environment becomes virtually local and

homogeneous to the client. The changes in object implementation, or in object relocation has no effect on the clients. Clients and servers are insulated from each other by freeing them from having low-level knowledge about what programming interface each supports.

CORBA Interfaces are defined in IDL, independent of any programming language. Components specify in IDL the types of services that they provide, including the methods they export and their parameters, attributes, error handlers and inheritance relationships with other components. IDL becomes the contract that binds clients to server components. In order to use or implement an interface, the interface must be translated into corresponding elements of a programming language. The translation mapping is done by the IDL compilers. These compilers are developed by the software vendors which support the CORBA standards. Compiling an IDL code produces client stubs and server skeletons. A client stub maps IDL operation definitions into procedural routines that are called to invoke a request. The server skeleton makes it possible for the ORB and an Object Adapter to translate the client request to a specific method on the server. In contrast to the client stubs, the dynamic invocation interface is unique for all object interfaces. The information about the operations invoked and types of the parameters can be obtained dynamically from the Interface Repository. On the server side, either skeletons or dynamic server interface are used to receive invocations on objects through the ORB and an Object Adapter. For each object implementation a separate implementation skeleton is generated and bound to the implementation code. IDL generated implementation skeleton provides the interface from ORB to the objects.

IDL is object-oriented, allowing abstraction of interface representations, polymorphic messaging, and inheritance of interfaces. IDL is similar to C++ language, but it is not a programming language, and does not support any operations, implementations, control structures and loops. It only defines the interface of the distributed object that is to be shared by clients and servers. Following is an example IDL definition.

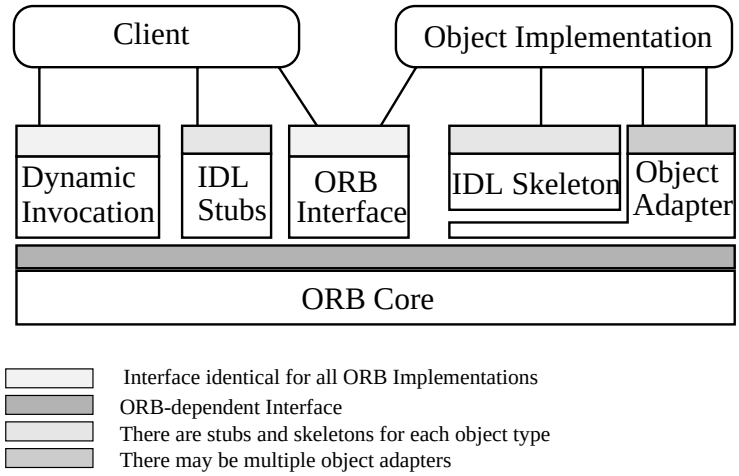


Figure 2.2: CORBA internal structure

```
exception DenyApproval {
    unsigned long reason;
};

interface Employee {
    attribute string employee_info;
    void promote(in char new_job);
    void transfer(in short new_dept);
};

interface Manager : Employee {
    void approve_transfer (
        in Employee employee_obj,
        in short current_department,
        in short new_department
    )
    raises DenyApproval;
};
```

In this example interfaces for two objects, *Employee* and *Manager* are defined. Employee object has got an attribute that represents information about an employee. Two methods are defined for an employee: promote to a new job and transfer to a new department. Manager is an employee which has got all the attributes and methods of an employee, therefore it can inherit the interface of employee object. Manager has additionally a method for approving the transfers of employees. The manager can give some reasons in the *DenyApproval* exception if s/he does not approve the transfer of an employee. The exceptions of an interface are raised in the implementation of the server. The exceptions are catched by the clients of the interface during run-time. The IDL code shall be compiled by a CORBA compiler which generates client stubs and server skeletons.

2.1.2 WfMC and Workflow Reference Model

Most of the available workflow management products have some common functionalities and characteristics because they aim at the same functional target. Until recently there has been no common standard for these products to make them interoperate with each other. In order to provide the communication and interoperation of workflows across different vendor products, a standard workflow specification is necessary. Workflow Management Coalition (WfMC), founded in 1993, is a non-profit, international organization of workflow vendors and analysts. Their objectives include standardization of the terminology and interoperability between workflow products.

A partial solution to semantic interoperability problem specific to workflow systems can be obtained by complying to the standards being developed by WfMC. WfMC defines a Workflow Reference Model [19] and interface descriptions between the basic components of the Reference Model. Figure 2.3 illustrates the major components and interfaces within workflow architecture of the model.

The core component in the Reference Model is the Workflow Enactment Service which is responsible for creation, management and execution of workflow process instances according to process definition produced by process definition tools. The workflow enactment software consists of one or more workflow engines,

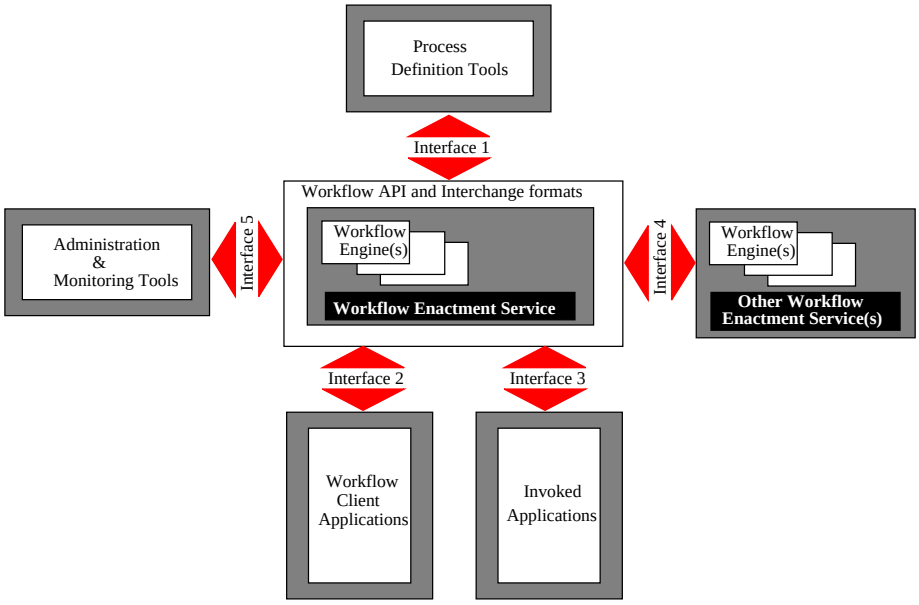


Figure 2.3: WfMC's Workflow Reference Model

which are responsible for managing all, or part of the execution of individual process instances. It interprets a process definition coming from the definition tool and coordinates the execution of workflow client applications for human tasks and invoked applications for computerized tasks. This may be set up as a centralized system with a single workflow engine responsible for managing all process execution or as a distributed system in which several engines cooperate, each managing part of the overall execution.

Process Definition Tools are used to analyse, model, describe and document a business process. The outcome of this process modeling and design activity is a process definition which can be interpreted at run time by the workflow engine(s) within the enactment service.

Workflow Client Applications involve the activities which require interaction with the human resources. In the workflow model, interaction occurs between the client application and a particular workflow engine through a well-defined interface embracing the concept of a worklist - the queue of work items assigned to a particular user by the workflow engine. Invoked Applications are the programs invoked by the workflow management system.

Administration and Monitoring Tools are used to track workflow process events during workflow process execution.

In order to achieve interoperability among various WFMS implementations, WfMC has defined five standard interfaces between the components. These interfaces are designated as Workflow APIs and interchange formats. These interfaces are :

Interface 1: Process Definition Tools Interface The purpose of this interface is to integrate process definitions generated by different process definition tools, or to use a definition generated for a workflow system in another system.

Interface 2: Workflow Client Applications Interface Users of a workflow system utilize several different types of client applications such as editors, CAD/CAM tools, WWW browsers. This interface provides integration of these applications in order to participate in a workflow system.

Interface 3: Invoked Applications Interface WFMSs are expected to work with already existing software components such as legacy systems and DBMS applications.

Interface 4: Other Workflow Enactment Services Interface It may be necessary for different enactment services to interoperate because a process in one enactment service may invoke a process in another.

Interface 5: Administration and Monitoring Tools Interface An administration and monitoring tool may be a separately developed system, or it may be necessary to centrally monitor different workflow systems.

2.2 Workflow Management Systems

The main task of a workflow management system is to schedule the execution of activities. In scheduling these activities, the WFMS determines what to execute next, locates the tools associated with each activity, transfers information

from activity to activity, assigns activities to users, checks the timeliness of the execution, monitors the overall progress and determines when the process has been completed successfully. The following sections describe some of the existing WFMSs.

2.2.1 ConTract

One of the first and most advanced workflow research projects is the ConTract project [40]. The focus of this project has been to extend transaction-oriented run-time mechanisms for fault tolerant workflow execution in a distributed environment.

The ConTract model tries to provide the formal basis for defining and controlling long-lived, complex computations, just like transactions control short computations. It is inspired by the mechanisms for managing flow that are provided by some TP-monitors, like queues and context-databases [10].

The basic idea of the ConTract model is to build large applications from short ACID transactions and to provide an application independent system service, which exercises control over them. As a main contribution, ConTracts provide the computation as a whole with the reliability and correctness properties. The ConTract Model extends the traditional transaction concepts to form a generalized control mechanism for long-lived activities. A large distributed application is being divided into a set of related processing steps which have appropriate consistency and reliability qualities for their execution.

A ConTract can be defined as consistent and fault tolerant execution of an arbitrary sequence of predefined actions (called steps) according to an explicitly specified control flow description (called script). In other words, a ConTract is a program that has control flow like any programming environment, has persistent local variables, accesses shared objects with application oriented synchronization mechanisms and has precise error semantics.

In ConTract programming model, the coding of steps is separated from defining an application's control flow script. Steps are coded without considering implementation details like managing asynchronous or parallel computations, com-

munication, resource distribution, synchronization and failure recovery.

Control flow between related steps can be modeled by the usual elements: sequence, branch, loop, and parallel constructors. *PARFOREACH(query)* statement executes its query in parallel. A sample ConTract *script* can be like:

```
CONTRACT Business_Trip_Reservations
CONTROL_FLOW_SCRIPT
 S1: Travel_Data_Input(in_context: ;out_context: date, from, to);
 PARFOREACH (airline: EXECSQL select airline from ... ENDSQL)
  S2: Check_Flight_Schedule(in_context: airline, date, from, to;
                            out_context:  flight_no, ticket_price);
 END_PARFOREACH
     ...
END_CONTROL_FLOW_SCRIPT
        ...
END_CONTRACT
```

Each of the *Sn* states is a *step* whose implementation can be coded in a programming language.

Each ConTract step is implemented by embedding it into a traditional ACID transaction, preserving only local consistency for the manipulated objects. Due to not being a transaction, a whole ConTract is not an ACID unit of work.

The ConTract script programmer can define atomic units of work consisting of more than one step by grouping them into sets. Furthermore, the transaction programmer may specify events depending on the result of steps and/or transactions. Grouping of a set of steps into one atomic unit of work is modeled by the following example:

```
TRANSACTIONS
  T1 ( S1, S2 )
  T2 ( S3, S4 )
END_TRANSACTIONS
```

Dependency between the execution results of different steps can also be modeled:

```
DEPENDENCY (T1 abort -> begin T1)
```

ConTract introduces concept of *context elements*, which are ConTract-local variables . These variables are kept in stable transactional storage and are only

accessible from the ConTract in which they are defined. Only a step of a ConTract can read or modify the value of *context elements*. Since the steps are running under the protection of a transaction, their modifications of the context are protected by this transaction. To support the examination of history, context elements are stored in an update-free way, in which a new version is generated upon a change to the variable.

APRICOTS (A PRototype Implementation of a COnTract System) [32] is an implementation of the ConTract model to show feasibility of its mechanism.

2.2.2 METEOR

METEOR (Managing End To End ORganization) [22] is a workflow language and model based on an extension of the Bellcore workflow model [30]. Its workflow execution model is driven by intertask dependency rules that are expressed in specifically designed script language. METEOR allows workflow definition at two levels of abstraction, by using two different languages: the Workflow Specification Language (WFSL), describing workflow structure and data exchange among tasks, and the Task Specification Language (TSL), describing the details of the tasks.

WFSL is used to specify the workflows, including all task types and classes in a workflow, all intertask dependencies, application level failure recovery and error handling issues. WFSL is a declarative rule-based language that mainly deals with task structure, typed input and outputs for each task, and preconditions for every controllable transition. The designer may define *task types*, describing task structures, and *task classes*, that are of a specific type and have typed input and outputs. Three task structures in this model are *transactional, non-transactional*, and *open 2PC task* structures. Each task has got *controllable* transitions that can be enabled by the workflow controller, and *non-controllable* transitions that are enabled by the task's processing entity. There are also *compound tasks* which can be composed of *simple tasks* and/or other compound tasks.

One of the key objectives of TSL is the minimal rewriting of existing tasks. TSL provides a wrapper for code describing interaction with an interface to pro-

cessing entity and essentially comprises a set of macros that can be embedded in a host language. The main functionality of the TSL macros is to indicate points in the task execution at which the workflow controller can be informed about the current logical state of the task. This functionality allows the workflow to deal with legacy applications without changing their code. Description of a simple transactional task can be :

```
simpleTaskType SIMPLE_TRANSACTIONAL
{
  CONTROLLABLE start(initial, executing) input ;
  NON_CONTROLLABLE abort (executing, aborted) output ;
  NON_CONTROLLABLE commit (executing, committed) output ;
}
```

simpleTaskType denotes that the definition does not relate a compound task; and also two non-controllable transitions produce output.

Each WFSL rule has two components: a control part and an optional data transfer part. Every next step of the workflow is determined by an evaluation of the relevant rules when an event occurs.

Intertask dependencies [2] determine how the tasks in the workflow are coordinated for execution. Two general types of dependencies in METEOR are *state dependencies* and *value dependencies*. State dependencies describe how a controllable transition of a task depends on the current observable states of other tasks. A *state dependency* example takes the following form:

[L1, done] ENABLES [L2, start]

This indicates that the start transition of L2 can be enabled only after L1 has entered the done state. This approach is similar to the transitions in ECA (Event-Condition-Action) rules, however ECA rules use events on the left hand in contrast to *states* in this case. Each state can be the result of one or more events being happened. An example of a rule including state and value dependencies is:

[L1, done] & (success(L1.output) = TRUE) ENABLES [L2, start]

success is a filter function which determines whether L1 has logically completed successfully. WFSL allows to associate output of a task with input of others. The specification:

L1.output → L2.input;

indicates that output of L1 must be used as input of L2.

2.2.3 INCAs: INformation CArriers

INCA model [3] proposes a workflow system with autonomous processing stations that are distributed, partially automated and partially connected to the network of other processing stations. The workflow control and execution is carried out as interactions between INCAs and processing stations. There is no central controlling mechanism and processing stations can be dynamically modified.

In this model, an INCA is associated with each workflow and contains information about private data and history of the activity and a set of rules. The set of rules defines the control and the data flow between the steps of the activity.

Nesting of workflows is possible by assuming that each processing stations may itself be a complex workflow that has to be carried out as an INCA computation. The INCA rules permit multiple children of the INCA computation to execute concurrently.

Compensating actions and Sagas [16] are used to ensure the failure atomicity of INCAs. The following rules are examples of using compensating actions in INCA rules:

```
On abort of step[i]
  Do execute inverse of step[i-1]
On commit of step[i]
  Do execute step[i+1]
```

The following two rules are associated with the compensating step of step[i]:

```
On abort of compensating step of step[i]
  Do execute compensating step of step[i]
On abort of compensating step of step[i]
  Do execute compensating step of step[i-1]
```

Durability of the computations can be achieved by making each successfully terminated step of the computation persistent and recoverable.

The implementation of the INCA model uses an INCA shell that is responsible for encapsulating an existing process station's software with a layer that

contains the minimum functionality necessary to support INCA computations. An INCA shell is composed of an agent and a rule management system. The agent is responsible for dealing with persistence of data, rules and execution results; context binding and step execution of the processing stations and directing the INCA rules to the appropriate destinations. The rule management system processes the rules brought by the INCA. INCA model adopts the ECA rules as the language. The events corresponding to INCA rules could be either *simple* events like state changes of the procedure, or *composite* events consisting of other simple events.

2.2.4 Exotica and FlowMark

Exotica Research Project [23, 1] being carried out at the IBM Almaden Research Center brings together industrial trends and research issues in the workflow area. It has got focus on a commercial product called *FlowMark* [14].

Exotica project has got six major research areas: failure resilience in distributed WFMSs, compensation and navigation in workflow networks, high availability through replication, mobile computing, distributed coordination and advanced transaction models.

The system, *Exotica/FMQM* [1], FlowMark on Message Queue Manager, is a distributed workflow system in which a set of autonomous nodes cooperate to complete the execution of a process. Each node functions independently of the rest of the system, the only interaction between nodes is through persistent messages informing that the execution of a step of the process has been completed. In this system, the sequence of events is as follows: a user first creates a process. The process is compiled in the process definition node. After compilation, the process is divided in several parts and each part is distributed to an appropriate node. The division of the process into parts will be based on the users associated with the different nodes and the roles associated with the different activities in the process. Upon receiving its part of the process, a node creates a process table to store this information and starts a process thread to handle the execution of instances of such process. Finally, the process thread creates a queue

for communicating with other nodes all information relevant to instances of the process.

The FlowMark workflow model is a representation of a process, comprising a process diagram and the settings that define the logic behind the components of the diagram. Listed below are the main components of a FlowMark workflow model:

Process: A sequence of activities that must be completed to accomplish a task.

Activity: A step within a process that represents a piece of work that the assigned person can complete by starting a program or another process.

Block: Grouping of several activities and nested blocks to reduce complexity and looping through a series of activities.

Control flow: Determines the sequence in which activities are executed.

Connector: Links activities in a workflow model to define the sequence of activities and the transmission of data between activities.

Data container: Allocated storage for the input and output data of the process and of the activities and blocks within it.

Data structure: An ordered list of variables with a name and a data type.

Condition: The means by which the flow of control in a process can be specified.

Program: A computer-based application program that supports the work to be done in an activity.

Server: A server can be specified for each subprocess, so that a process can be distributed among several servers.

Staff: Each activity in a process is assigned to one or more staff members.

FlowMark provides a graphical definition tool that can be used to model a workflow, by including the above components. There are symbols for activities and blocks, and a process can contain many of these blocks and activities.

The information given in a graphical form can be exported to a textual form. This definition contains declarations for data structures, programs, servers and staff. The process definition gives some information about the process and how the workflow is supposed to execute.

2.2.5 OpenPM and AdminFlow

The OpenPM prototype was designed by HP as an open, enterprise capable, object-oriented workflow system to manage business activities supporting complex enterprise process in a distributed heterogeneous computing environment [33]. It is a middleware service that represents a substantial evolution from first generation workflow technologies and forms the base of HP AdminFlow product.

OpenPM provides a generic framework and a complete set of services for business process flow management with emphasis on performance, scalability, availability and system robustness. Basically, it provides:

- an open system adhering to CORBA-based communication infrastructure and providing WfMC standard interface.
- high performance due to optimized database access and commitment.
- effective management due to OpenView-based system management environment.
- a total solution for business re-engineering including a complete set of business application and application development tools.

The core component of the OpenPM architecture is the OpenPM engine which supports five interfaces for business process defining, business process execution, business process monitoring, resource and policy management, and business object management. A business process is defined via the process definition interface. An instance of the business process can be started, stopped, or intervened via the process execution interface. Status information of each process instance, configuration and load information of the entire system can be queried via the process monitoring interface. The resource and policy management interface is

used to allocate, at run time, execution resource to a task according to the policies defined by the organization and availability of resources. The execution of business activities is performed via the business object management interface.

In OpenPM, a business process is represented as a directed graph comprising a set of nodes connected by arcs. There are two kinds of nodes: work nodes and rule nodes, as well as two kinds of arcs: forward arcs and reset arcs. Work nodes represent activities to be performed external to the OpenPM engine. Rule nodes represent processing internal to the OpenPM engine. This processing includes decisions of what nodes should next execute, the generation or reception of events, and simple data manipulation. A rule language is used to program the rule node decision. Forward arcs represent the normal execution flow of process activities and reset arcs are used to support repetitions or explore alternatives in a business process.

The OpenPM engine launches business process instances in response to user requests. For each instance, OpenPM engine steps through the nodes according to the order specified in its business process definition. The engine interacts with business activities supported by various kinds of implementations ranging from manual handling by human to automated application execution by computer. Based on CORBA technology, in OpenPM, an abstraction, called business object, is built to encapsulate whatever the piece of work each process activity has to accomplish.

2.3 Previous Work on Workflow Enactment Service

Previous work on scheduling of workflow tasks mostly concentrates on the transactional behavior of the workflow management systems. For this purpose several extended transaction models have been proposed in the literature [13, 31, 34]. As a general approach, their transaction model semantics are directly embedded to the enactment service. However, they either suffer from the formal definition of execution model or they do not provide distributed scheduling of tasks. This results in lack of interoperability among WFMSs, inadequate performance for some business processes, lack of support for correctness and reliability and finally, weak

tool support for analysis, testing, and debugging workflows.

In general, three methods have been used for the implementation of workflow enactment service :

- State and activity chart based approach
- Knowledge Base (ECA Rules) approach
- Petri-Net based approach

An example of the state and activity chart based approach is described in the Mentor project [35, 41, 42]. It uses the formalism of state and activity charts and a commercial tool, Statemate, for workflow specification. A method is proposed for transforming a centralized state chart specification into a form that is amenable to a distributed execution and to incorporate the necessary synchronization between different processing entities. Although a concrete formalism is introduced to the workflow specification in this project, transformation process requires a human intervention and does not guarantee the most efficient distributed execution.

In Meteor [22] task structures are used for the representation of the execution behavior of the tasks. Each task structure has an initial state, and on the start transition moves to the executing state. Two types of tasks are specified: Simple tasks and compound tasks. A simple task is a physical unit of work that executes at a processing entity. Compound tasks are logical units of work that are not executed against processing entities, but mean to specify co-ordination and data flow requirements between sub-tasks. A workflow can itself be specified by as a compound task. Meteor execution engine uses Event-Condition-Action (ECA) rules for defining the preconditions for each controllable transition in each task. Specification of a workflow is a very complex process in Meteor and it lacks the formal definition of execution engine. In addition, scheduler is not distributed.

Petri-Nets is a model representation, often in mathematical terms, of the important features of the object or system under study. The execution of a Petri-Net is controlled by the number and distribution of tokens in the net. Tokens reside in the places and control execution of the transitions in the net. A Petri-Net executes by firing transitions. A transition fires by removing tokens its input

places and creating new tokens which are distributed to its output places. Petri-Nets are indeed an important formalism in which to express computations and can be applied to workflows [43]. However, they are too complex and include redundant characterizations for the specification of the workflows.

[10] describes an execution model, based on an extended nested transaction model, to govern the concurrency and recovery properties of transactions with triggered ECA rules. The model supports arbitrary events, not just database updates. However, this approach does not address any solution for the distributed computing and it is not practical. As a continuing research of [10], [11] proposes a new model namely ATM, which is a transactional model of activities that is based on an extended nested transaction model. This extended model provides greater flexibility in specifying the scope of execution of a nested transaction. Critical activity concept has been introduced in this work. Execution model allows the activities to consist recursively of steps that may be subactivities or transactions. The model defines precisely the semantics of activities. Although our workflow definition model seems to be very similar to this approach, we do not base our system to ECA rules and a different transaction model is utilized. [4] further extends these works by proposing an extensive transaction model and a technique for handling failures. This transaction model has in-process open nesting for extending closed/open nesting to accommodate applications that require improved process-wide concurrency without sacrificing top level atomicity. In this model a process consists of hierarchically structured activities. An activity represents a logical piece of work that contributes to a process. In addition, a formalism to the nested activity modeling is introduced. Although their failure handling approach is very effective, its implementation is not possible with the current database products.

The ConTract Model [29, 40] tries to provide the formal basis for defining and controlling long-lived, complex computations, just like transactions control short computations. The basic idea is to build large applications from short ACID transactions and to provide an application independent system service, which exercises control over them. As a main contribution, ConTracts provide the

computation as a whole with reliability and correctness properties. A ConTract is defined as a consistent and fault tolerant execution of an arbitrary sequence of predefined actions (called steps) according to an explicitly specified control flow description (called scripts). The design rationale behind ConTract model is the objective to capture system failures by the system and to present the user or application programmer with an arbitrarily reliable execution platform.

[15] demonstrates a general purpose programming language that is extended to serve workflow management requirements. This paper proposes that there is no need to develop yet another language for the workflow specification. Workflow language requirements are specified and an extension to C language is explained. Communication variables are used to provide concurrency and information exchange between processes. Since a communication variable can be assigned a value only once, implementation of such a system may result into prohibitive amount of communication variables. Moreover, this paper does neither address the problems related to architectural design of WFMSs nor provide any formal foundation to the workflow activity scheduling.

ACTA Formalism [5, 6, 7, 8] is the first serious attempt to provide a formal framework for the transaction models. ACTA allows the specification of the effects of transactions on other transactions and also their effects on objects. Inter-transaction dependencies form the basis for the former while the visibility of and conflicts between operations on objects form the basis for the latter. ACTA captures the extended functionality of a transaction model (1) by allowing the specification of significant events beyond commit and abort, (2) by allowing the specification of arbitrary transaction structures in terms of dependencies involving any significant events, (3) by supporting finer grain visibility for objects in the database by associating a view and a conflict set with each transaction and the notion of delegation, (4) and by facilitating object-specific and transaction-specific semantic-based concurrency control. This framework is also utilized in our approach to separate scheduler and the transaction model semantics. Therefore, our workflow scheduler approach is transaction model independent.

[2] constructs finite automata for dependencies. It uses pathset search to avoid

generating product automata, but the individual automata can be quite large. Therefore, complex dependencies can not be expressed or processed easily with this approach. Moreover, it is centralized.

In general, these approaches are criticized for their inadequate power of representing the business modeling, their complexity, their implementation difficulty, inflexibility and unavailability to the distributed execution. Recently a new approach called Event-Based Approach is proposed by [36, 37]. The approach taken in [36, 37] for distributed scheduling of workflow executions is based on the following observations and mechanisms: The main activity of a workflow is organization and coordination of tasks which might be dependent on each other by their states. For example start of one task may depend on the commitment of another task. Since state changes of the tasks are represented as event generations, state dependencies can be defined through event ordering. Therefore, controlling the occurrences of events provides the coordination of the tasks. In other words, intertask dependencies are represented by the event dependencies. To make distributed execution of workflow computations possible, occurrences of events are not controlled in a central scheduler. Instead, each event is made responsible for controlling its execution to decide on the right time to occur. Required information for this operation is obtained from the dependency expressions after a refinement process which is termed as guard compilation. Guards are temporal expressions defined on events and occurrences of events are permitted only if their guards are true. When an event happens, messages announcing its occurrence are sent to other related events so that the effects of the occurrence of that event are reflected to the whole system. Tasks are interfaced to the system through agents. An agent embodies a coarse description of the task, including states and transitions. In addition, an actor is instantiated for each event. The actor for an event maintains its current guard and manages communication of necessary messages. Guards of events are determined by generating all possible computations relevant to each dependency. These computations are checked whether they satisfy the given dependency. Then guards considering only one dependency are constructed by using only the acceptable computations. Finally guards due to workflow are

obtained by combining the guards due to dependencies. In evaluating a guard, the intrinsic attributes of events must be taken into account. The following event attributes are defined in [37]: (a) Normal events that are delayable and rejectable (e.g. commit), (b) Inevitable events that are delayable and nonrejectable, (c) Immediate events that are nondelayable and nonrejectable (e.g. abort), and (d) Triggerable events that are forcible (e.g. start).

The following points have been notified with this approach:

1. In [2], guard generation process is said to run into combinatorial explosion. The proposed process first determines all possible paths for a given dependency. There are n! number of paths for a dependency involving n events. Later in [37], it is proposed that by relaxing the past and the future, the guards can be generated symbolically without the need for determining all possible paths.

2. The execution mechanism is based on message exchange between actors since guards on events require notification messages to assimilate the event executions. This in turn might arise potential race conditions and deadlocks. For example there could be two events waiting for the occurrence of each other, resulting in a deadlock. It is therefore essential to preprocess the guards so as to detect and resolve the potential deadlocks through promissory messages [37].

3. The guards may contain events that refer to the future, however in actual execution we do not have the luxury of looking into the future. In [37], various heuristics are suggested, yet the completeness of the suggested heuristics is left as a future work.

CHAPTER 3

METUFlow ARCHITECTURE

METUFlow is a distributed workflow management system prototype being developed at METU [12]. A simplified architecture of METUFlow system is given in Figure 3.1. In METUFlow, first a workflow is specified using a graphical workflow specification tool which generates the textual workflow definition in MFDL (METUFlow Definition Language) as will be explained in Section 3.1. The core component of a workflow management system is the workflow scheduler which instantiates workflows according to the workflow specification and controls correct execution of activities interacting with users via worklists and invoking applications as necessary. In METUFlow, the functionality of the scheduler is distributed to a number of guard handlers which contain the guard expressions for the significant events of the activity instances as explained in Chapter 4. Also, there exists a task handler which acts as an interface between the activity instance and its guard handler. Details of task handling in METUFlow are discussed in [20]. In a workflow management system, there may be activities in which human interactions are necessary. In METUFlow, work item scheduler manages such interactions. It is responsible for progressing work requiring user attention and interacts with the scheduler through user task handler as shown in Figure 3.1. Work item scheduler uses the authorization service to determine the authorized roles and users. The detailed architecture of work item scheduler is provided in Section 3.2. History manager (see Section 3.3) provides the mechanisms for

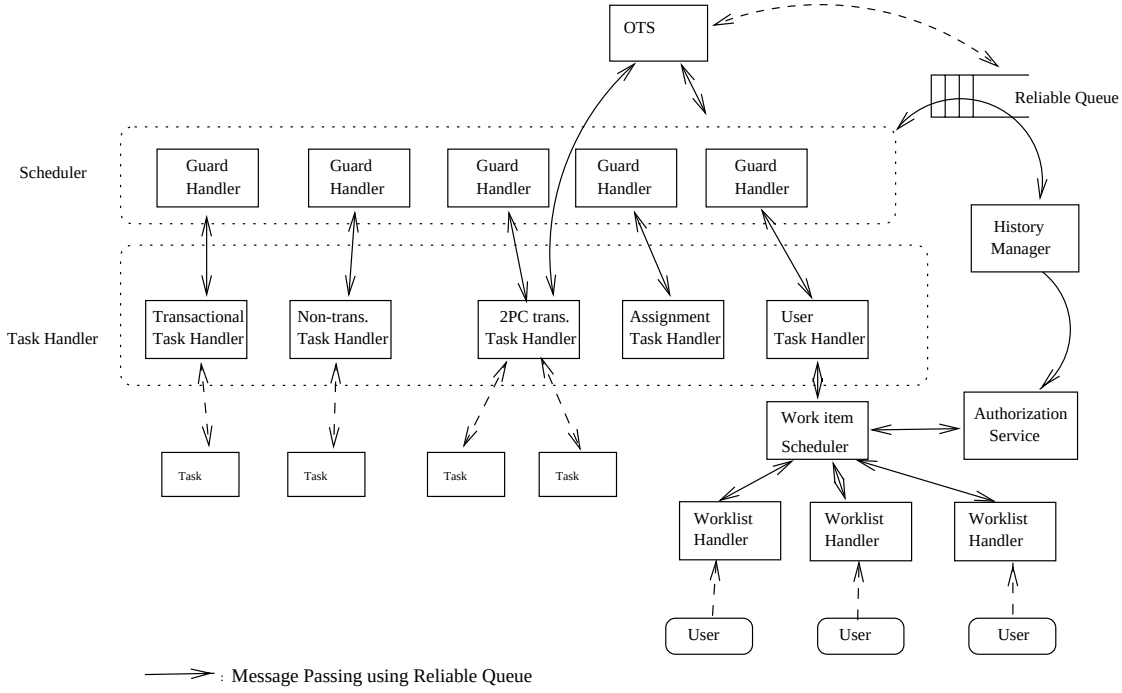


Figure 3.1: The simplified architecture of METUFlow

storing and querying the history of both ongoing and past processes. It communicates with the scheduler through a reliable message queue to keep track of the execution of processes. The communication infrastructure of METUFlow is CORBA. CORBA does not provide for reliable message passing, that is, when ORB crashes, all of the transient messages are lost. For this reason, we have implemented a reliable message passing mechanism which uses Object Transaction Service (OTS) based transaction manager (see Section 3.4) to commit distributed transactions. Note that reliable message passing is necessary among all the components of METUFlow such as between guard handlers and task handlers as indicated in Figure 3.1. To use CORBA, a CORBA interface must be defined for each of the distributed components in CORBA IDL.

3.1 The Process Model and The METUFlow Definition Language

We have found out that introducing block structures and thus confining the inter task dependencies to a well-structured form provides great ease and efficiency in

guard generation process [39]. Thus a workflow process is defined as a collection of blocks, tasks and other subprocesses. A task is the simplest unit of execution. Processes and tasks have input and output parameters corresponding to workflow relevant data to communicate with other processes and tasks. We use the term activity to refer to a block, a task or a (sub)process. Blocks differ from tasks and processes in that they are conceptual activities which are present only to specify the ordering and the dependencies between activities.

We have defined eight types of blocks, namely, serial, and_parallel, or_parallel, xor_parallel, contingency, conditional, iterative and for_each blocks [12, 17, 39]. The following definitions describe the semantics of the block types, compensation activity and undo task where A stands for an activity (block, task or process) and B for a block.

Serial Block B = $(A_1; A_2; A_3; .....; A_n)$: Start of a serial block B causes A_1 to start. Commitment of A_1 causes start of A_2 and commitment of A_2 causes start of A_3, and so on. Commitment of A_n causes commitment of B. If one of the activities aborts, the block aborts. If the block aborts, its committed activities should be compensated in the reverse order.

And_Parallel Block B = $(A_1$ & A_2 & ..... & $A_n)$: Start of an and_parallel block B causes start of all of the activities in the block in parallel. B commits only if all of the activities commit. If one of the activities aborts, the block aborts. If the block aborts, its committed activities should be compensated in parallel.

Or_Parallel Block B = $(A_1|A_2|.....|A_n)$: Start of an or_parallel block B causes start of all of the activities in the block in parallel. At least one of the activities should commit for B to commit but B can not commit until all of the activities terminate. B aborts if all the activities abort. If B aborts, its committed activities should be compensated in parallel.

Xor_Parallel Block B = $(A_1||A_2||.....||A_n)$: Start of an xor_parallel block B causes start of all tasks in the block in parallel. B commits if one of the

activities commits, and commitment of one activity causes other activities to abort. If all of the activities abort, the block aborts.

Contingency Block B = $(A_1, A_2, ...A_n)$: Start of a contingency block B causes start of A_1. Abort of A_1 causes start of A_2 and abort of A_2 causes start of A_3,and so on. Commitment of any activity causes commitment of B. If the last activity A_n aborts, the block aborts.

Conditional Block B = (condition, A_1, A_2): Conditional block B has two activities and a condition. If the condition is true when B starts, then the first activity starts. Otherwise, the other activity starts. The commitment of the block is dependent on the commitment of the chosen activity. If the chosen activity aborts, then B aborts.

Iterative Block B = (condition; $A_1; A_2; .....; A_n$): The iterative block B is similar to serial block, but start of iterative block depends on the given condition as in a while loop and execution continues until either the condition becomes false or any of the activities aborts. If B starts and the condition is true, then A_1 starts and continues like serial block. If A_n commits, then the condition is reevaluated. If it is false, then B commits. If it is true, then A_1 starts executing again. If one of the activities aborts at any one of the iterations, B aborts. If B aborts, its committed activities for all the iterations should be compensated in the reverse order.

For_Each Block B = (list; par_type; B_1): Everything inside a for_each block is considered as a serial block named B_1. The list is an array variable on each element of which this serial block will be applied in a parallel way specified with the par_type parameter. Start of a for_each block causes the start of B_1 for each element of the list in parallel. Commit of B is dependent on the value of the par_type. Par_type can be and_parallel, or_parallel or xor_parallel. For example, if par_type is and_parallel, then B commits only if all started B_1 blocks have committed and B aborts if one of the started B_1 blocks has aborted. This means that after B_1 is generated for each element

of the list, B behaves like one of the parallel block types according to the value of the par_type. The purpose lying behind this block is to provide efficiency by executing the same actions on individual items in a list in a parallel fashion.

Compensation Activity A = A_c, where A_c is the compensation activity of A. The compensation activity A_c of A starts if A has committed and should be cancelled due to failures of other activities in the process like the abort of some antecedent activity. If both an activity and its subactivities have compensation, only the compensation of the activity is used.

Undo Task T = T_u, where T_u is the undo task of task T. The undo task T_u of T starts if T fails.

We have implemented a specification language based on these structures, called METUFlow Definition Language (MFDL), within the scope of the METUFlow project. In MFDL, the tasks involved in a business process, the execution and data dependencies between these tasks are provided. The WfMC has identified a set of six primitives to describe flows and hence construct a workflow specification [19]. With these primitives, it is possible to model any workflow that is likely to occur. These primitives are: sequential, AND-split, AND-join, OR-split, OR-join and repeatable task. These primitives are all supported by MFDL through its block types. Of the above block types, serial block implements the sequential primitive. And_parallel block models the AND-split and AND-join primitives. AND-split, OR- join pair is modelled by or_parallel block. Conditional block corresponds to OR-split and OR-join primitives. Finally, repeatable task primitive is supported by the iterative block. For_each block can be represented as one of the parallel block types. But, here the number of blocks executing in parallel can only be decided at run-time instead of compile-time.

The following is an example of a workflow defined in MFDL:

```
DEFINE_PROCESS manufacture()
{
  VAR int product_no, quantity, order_no, customer_id;
  VAR date due_date;
```

```
    VAR part_list part_no_list;
    VAR index_list i;

    get_order (OUT product_no, OUT quantity, OUT due_date,
               OUT order_no, OUT customer_id);
    enter_order (IN product_no, IN quantity, IN due_date,
                 IN order_no)
                COMPENSATED_BY delete_order (IN order_no);
    check_bill_of_material (IN product_no, OUT part_no_list);
    FOR_EACH (part_no_list, PAR_AND)
    {
        check_stock (IN part_no_list[INDEX].part_no,
                     IN part_no_list[INDEX].part_quantity,
                     OUT part_no_list[INDEX].status,
                     OUT part_no_list[INDEX].raw_mat_no,
                     OUT part_no_list[INDEX].raw_mat_quant,
                     OUT part_no_list[INDEX].part_quant_to_prod);
        IF (part_no_list[INDEX].status == 0) THEN // No_Stock
           vendor_order (IN part_no_list[INDEX].raw_mat_no,
                         IN part_no_list[INDEX].raw_mat_quant);
        withdraw_from_stock (IN part_no_list[INDEX].raw_mat_no,
                             IN part_no_list[INDEX].raw_mat_quant)
        COMPENSATED_BY add_to_stock
                      (IN part_no_list[INDEX].raw_mat_no,
                       IN part_no_list[INDEX].raw_mat_quant);
        get_process_plan (IN part_no_list[INDEX].part_no,
                          OUT part_no_list[INDEX].process_plan,
                          OUT part_no_list[INDEX].no_of_steps);
        i[INDEX] = 0;
        WHILE (i[INDEX] < part_no_list[INDEX].no_of_steps) DO
        {
               produce (IN part_no_list[INDEX].cell_id,
                        IN part_no_list[INDEX].part_quant_to_prod,
                        IN order_no,
                        IN part_no_list[INDEX].raw_mat_no,
                        IN part_no_list[INDEX].raw_mat_quant);
               i[INDEX] = i[INDEX] + 1;
        }
     }
     assemble_product (IN product_no);
  }
```

The above example is a simplified workflow of a manufacturing process carried out in a manufacturing company. First, a customer orders a product by the *get_order* task. Then, the order information is entered to the database. Bill of

material is checked to determine which parts are required to produce the ordered product. For each part in the bill of material, *check_stock* task is initiated to check stock database for the availability of the raw materials to manufacture that part. If any of the raw materials is missing in the stock, it is ordered from the vendors by the *vendor_order* task. Then, required raw materials to produce the part are withdrawn from the stock. The process plan is read from the database for the part in discussion. The raw materials are processed into the part by following the steps in the process plan using the *produce* subprocess. After all these activities are done in an and_parallel fashion for all of the parts, then the individual parts are assembled into the ordered product by the *assemble_product* subprocess. The reason why we use a for_each block is to speed up the process by executing the same operations for each of the parts in parallel.

MFDL provides constructs for naming workflow relevant data types, associating an identifier with a type. Basic types in MFDL are floating point, integer, character, string and object. In addition to these simple types, MFDL provides two constructed types: *structure* and *array*. CORBA supports the basic types but not structure and array types. Therefore, we handled the complex types separately.

In MFDL, we have used five types of tasks. These are TRANSACTIONAL, NON_TRANSACTIONAL, NON_TRANSACTIONAL with CHECKPOINT, USER and 2PC_TRANSACTIONAL activities. USER activities are in fact NON-TRANSACTIONAL activities. They are specified separately in order to be used by the worklist manager which handles the user-involved activities. The states and transitions between these states for each of the activity types are demonstrated in Figure 3.2. The significant events in our model are start, commit and abort. The event attributes of these tasks are shown in Table 3.1.

Note that the abort event of a 2PC_transactional task after the coordinator has taken a decision is normal whereas it is immediate before the coordinator has taken a decision. Triggerable and normal events are controllable because they can be triggered, rejected or delayed while immediate events are uncontrollable. We have chosen to include a second type of non_transactional activity, namely,

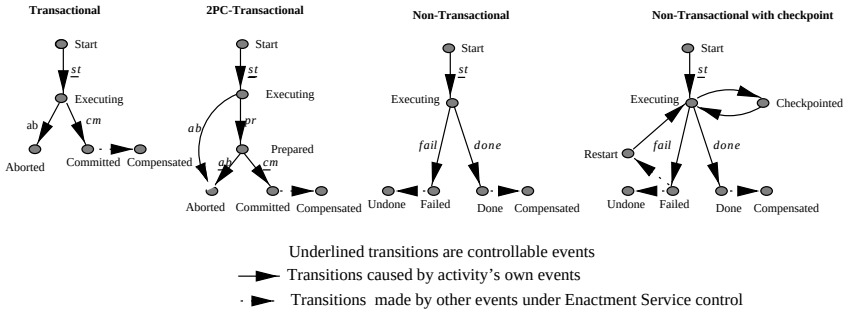


Figure 3.2: Typical task structures

Table 3.1: Event attributes

activity_types	start	abort/fail	commit/done
transactional	triggerable	immediate	normal
2PC_transactional	triggerable	normal,immediate	normal
non_transactional	triggerable	immediate	immediate
non_transactional with checkpoint	triggerable	immediate	immediate

NON_TRANSACTIONAL with CHECKPOINT, in our model by making the observation that certain non_transactional activities in real life, take checkpoints so that when a failure occurs, an application program rolls the activity back to the last successful checkpoint.

These activity types may have some attributes such as CRITICAL, NON_VITAL and CRITICAL_NONVITAL. Critical activities can not be compensated and the failure of a non_vital activity is ignored [4, 11]. Besides these attributes, activities can also have some properties like retriable, compensable, and undoable. A retriable activity restarts execution depending on some condition when it fails. Compensation is used in undoing the visible effects of activities after they are committed. Effects of an undoable activity can be removed depending on some condition in case of failures. Some of these properties are special to specific activity types. Undo conditions and activities are only defined for non_transactional tasks, because transactional tasks do not leave any effects when they abort. Only 2PC_transactional activities can be defined as critical. Note that the effects of critical activities are visible to the other activities in the workflow but the commitment of these activities are delayed till the successful termination of the workflow.

An activity can be both critical and non_vital at the same time, but can not be critical and compensable.

In MFDL, activities in a process are declared using the reserved word ACTIVITY. This declaration allows the sharing of an activity definition among many workflow processes with possibly different attributes and properties for each instance. Similarly, the variables are declared using the VAR reserved word.

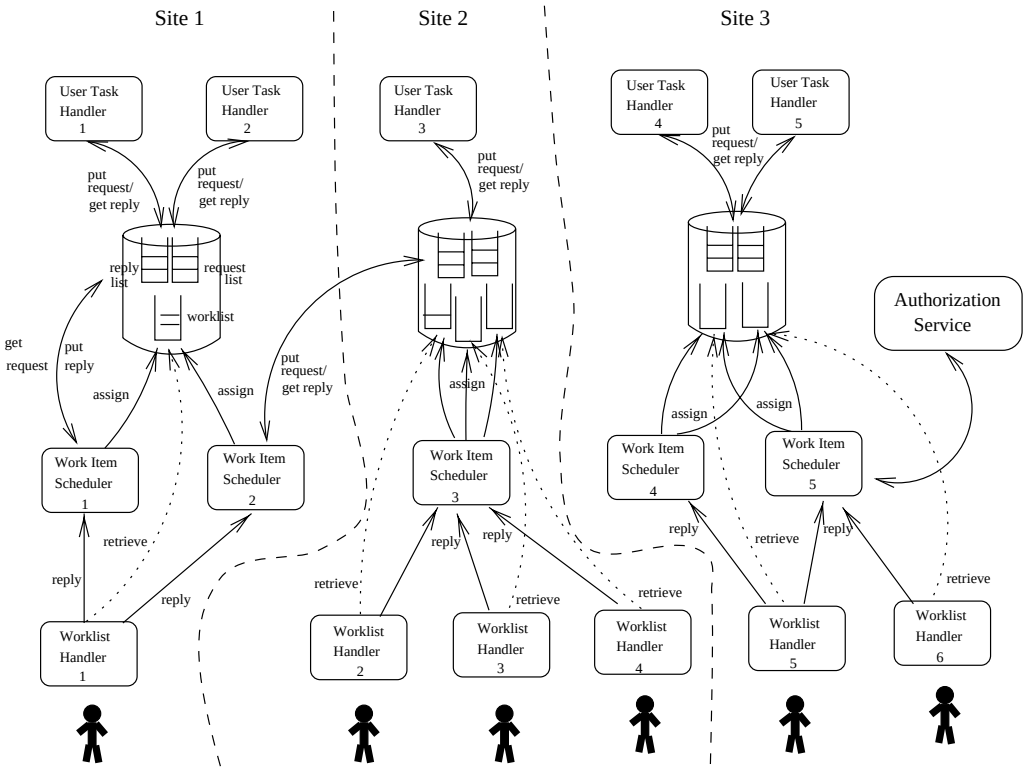
The full Backus-Naur Form (BNF) form of MFDL is given in Appendix A.

3.2 Worklist Management in METUFlow

The worklist manager is a software component which manages the interaction between workflow participants and the scheduler.

In METUFlow, the worklists are distributed, that is, a worklist at a site contains the work items to be accessed by the users at that site.

When a user activity is to be invoked by the scheduler, a user task handler created for this purpose stores the request (work item) into a request list within the scope of a transaction. Request list is a CORBA object and its implementation in a particular site depends on the persistent storage available in that site, that is, this CORBA object is implemented on a DBMS if it is available, otherwise it is implemented as a file. Worklist manager, as depicted in Figure 3.3 consists of two components. The first one, work item scheduler, decides on the assignment of work items to the worklists of the users in cooperation with the authorization service. The first version of the authorization service implemented contains the definitions of roles and their members, authorizations to execute tasks and constraints controlling the execution of these authorizations. We plan to improve this service by including periodic, temporal, event based distributed constrained authorizations and authentication services. The work item scheduler is also responsible for putting the reply back into the reply list, again, within the scope of a transaction. That is, the reply list is a persistent CORBA object whose implementation is realized through a DBMS or a file depending on the capabilities of the site concerned. The second component, worklist handler is responsible for retrieving work items to be presented to the user for processing.



retrieve	:	retrives the worklist contents for the presentation to the user
put/get_request	:	puts (gets) a work item into (from) request list
put/get_reply	:	puts (gets) reply from the user into (from) reply list
reply	:	sends reply back (from worklist handler to work item scheduler)
assign	:	inserts work items into the worklists of the user

Figure 3.3: Worklist manager in METUFlow

A point to be noted over here is the following: CORBA provides location transparency, in other words the users need not be aware of the location of the objects to be created. However, CORBA does provide mechanisms to affect the object creation site although the specifics depends on the ORB at hand. First by default, an object is created at the local site if it is possible. Therefore, whenever there is a request to create a work item scheduler, it is created at the same site with the user task handler. In order to be able to create worklists at the same host with the involved user (or role), a list is kept which stores the association between the user-ids and host-ids. In METUFlow, lookup method of Orbix's locator class [27] is used for this purpose.

Finally, in order to provide access to the worklists through World-Wide-Web,

we have chosen to implement them in Java which made it easier to connect to a CORBA compliant ORB, namely Orbix, through OrbixWeb.

3.3 History Management in METUFlow

Workflow history management provides the mechanisms for storing and querying the history of both ongoing and past processes. This serves two purposes: First, during the execution of a workflow, the need may arise for looking up some piece of information in the process history, for example, to figure out who else has already been concerned with the workflow at what time in what role. This kind of information contributes to more transparency, flexibility and overall work quality. Second, aggregating and mining the histories of all workflows over a longer time period forms the basis for analyzing and assessing the efficiency, accuracy and the timeliness of the enterprise's business processes. Therefore, this information provides the feedback for continuous business processes re-engineering. Given that, much of the history information relates to the time dimension in that it refers to turnaround times, deadlines, delays. over a long time horizon.

In consistency with its architecture, METUFlow history and workflow relevant data handling mechanism is based on CORBA. The history of each activity instance is implemented as a CORBA object. To exploit the advantages brought by the distributed execution of the workflow scheduler, history management should also be distributed. To make distributed history management possible, the persistent store in which the history information is kept, should also be distributed over the network.

The history of each activity instance is implemented as a CORBA object at the same site at which the activity object itself is invoked. If a DBMS is available at the concerned site, it is used as the persistent store, otherwise a binary file is used for this purpose. It is possible to have history objects created at the same site where they are activated to prevent the communication cost with the activity instance objects.

Each activity instance is responsible for its own history object and knows the object identifier of its parent activity instance. A child activity instance invokes a

method to pass the object identifier of its own history object to its parent object. A parent activity instance object establishes the links between its own history object and its child's history object. Note that in the eventual history tree of the process instance obtained this way objects are linked through their object identifiers according to the process tree.

When it comes to querying history both for monitoring and for data mining purposes, encapsulating these data as CORBA objects naturally yields to using the Query Service Specification of OMG.

The Query Service provides query operations on collection of objects. The Query Service can be used to return collections of objects that may be:

- selected from source collections based on whether their member objects satisfy a given predicate.
- produced by query evaluators based on the evaluation of a given predicate.

In summary, with a distributed history and workflow relevant data handling mechanism, availability and scalability aspects of the system are increased.

3.4 OTS-based Transaction Manager

In METUFlow, distributed transaction management is realized through a transaction manager that implements Object Transaction Service Specification of OMG, OTS [9, 25]. OTS Specification describes a service that supports flat and nested transactions in a distributed heterogeneous environment. It defines interfaces that allow multiple, distributed objects to cooperate, to provide atomicity of transactions. These interfaces enable the objects either to commit or rollback all the changes together in the presence of failure.

Figure 3.4 illustrates the major components and the interfaces defined by OTS. In a typical scenario, a transactional client (transaction originator) creates a transaction obtaining a Control object from a Factory provided by ORB. Transaction clients use the Current pseudo-object to begin a transaction, which becomes associated with the transaction originator's thread. The Current interface defines operations that allow a client of OTS to begin and end transactions

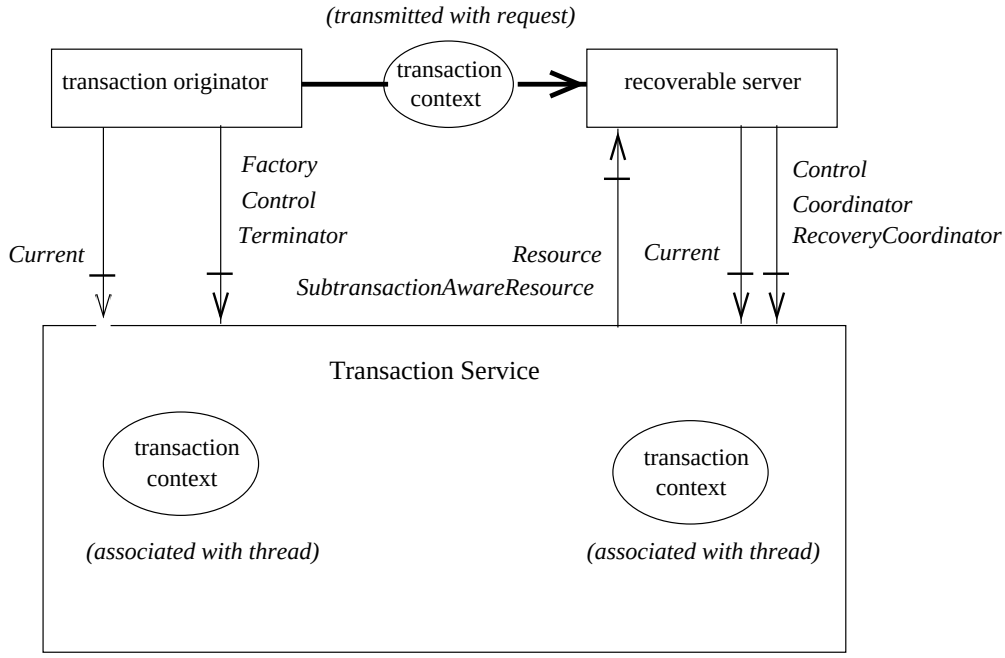


Figure 3.4: The major components and interfaces of the OTS

and to obtain information about the current transaction. A simplified version of Current interface is illustrated below:

```
interface Current {
 void begin();
 void commit();
 void rollback();
 Status get_status();
 string get_transaction_name();
 ...
}
```

ORB associates a transaction context with each Control object. A transaction context contains all the necessary information to control and to coordinate transactions. Transaction context is either explicitly passed as a parameter of the requests, or implicitly propagated by ORB, among the related transactional objects. The Control object is used in obtaining Terminator and Coordinator objects. Transactional client uses the Terminator to abort or to commit the transaction. Coordinator provides an interface for transactional objects to participate in Two Phase Commit (2PC) protocol. Transactional client sends requests to

transactional objects. When a request is issued to a transactional object, the transaction context associated with the invoking thread is automatically propagated to the thread executing the method of target object. A transactional object is the one that supports transaction primitives as defined by the standard. After the computations involved in the transaction have been completed, the transactional client uses the Current pseudo object to request that the changes be committed. OTS commits the transaction using 2PC protocol wherein a series of requests are issued to the registered resources. Thus, ORB provides the atomicity of distributed transactions.

In addition to the above usage of OTS, in METUFlow OTS implementation, a method is added to the Coordinator object to handle xor_parallel block which requires one and only one task to commit, for the commitment of the block.

CHAPTER 4

GUARD GENERATION

In METUFlow, the execution environment is fully distributed. Since the system is distributed on the basis of activities, each activity should know when its significant events like start, abort and commit should occur without consulting to a top-level central decision mechanism. For this purpose, we use temporal expressions which define the conditions under which an event should occur, called guards.

In this chapter, first how intertask dependencies are incorporated into guard expressions are explained. Then, the guard generation process is described in Section 4.2. After the guards are generated, their run time control is handled by the guard handler, which is summarized in Section 4.3. Finally, in Section 4.4, how execution of underlying tasks are handled is described.

4.1 Incorporation of Intertask Dependencies into Guards

At compile time of the process specification, it is possible to construct the guards for each of the activities by using the block structures which implicitly provide the necessary intertask dependencies. We use the ACTA formalism with slight modifications to express the semantics of the block structures[1]. ACTA is a framework for formal specification and analysis of transaction models that allows one

[1] We treat fail/done event of non_transactional activities as abort/commit of transactional activities.

to specify the effects of transactions on other transactions and also their effects on objects [5]. Following are the ACTA dependencies [8] that exist in our workflow system:

Let t_i and t_j be two transactions.

- **Commit Dependency**(t_j CD t_i): if transaction t_i commits, then t_j commits.
- **Commit-on-Abort Dependency**(t_j CAD t_i): if transaction t_i aborts, then t_j commits.
- **Abort Dependency**(t_j AD t_i): if transaction t_i aborts, then t_j aborts.
- **Abort-on-Commit Dependency**(t_j ACD t_i): if transaction t_i commits, then t_j aborts.
- **Begin Dependency**(t_j BD t_i): if transaction t_i begins executing, then t_j starts.
- **Begin-on-Commit Dependency**(t_j BCD t_i): if transaction t_i commits, then t_j begins executing.
- **Begin-on-Abort Dependency**(t_j BAD t_i): if transaction t_i aborts, then t_j begins executing.

We have added conditional dependencies to the ACTA dependencies. These dependencies have an additional argument which is "condition". For example, we express conditional begin dependency as BD(C). If condition C is true, then BD holds, else it does not hold.

Using these dependencies, we can formally restate semantics of block types, undo task and compensation activity defined in the previous chapter as:

Semantics 1 B = $(A_1; A_2; A_3; .....; A_n)$, where B is a serial block.

- A_1 BD B
- A_{i+1} BCD A_i , 1 $\leq$i<n

- B CD A_n
- B AD A_i ,1 $\leq$ i $\leq$n

Semantics 2 B = (A_1 & A_2 & ..... & A_n), where B is an and_parallel block.

- A_i BD B ,1 $\leq$ i $\leq$n
- B AD A_i ,1 $\leq$ i $\leq$n
- $\forall$i(B CD A_i)

Semantics 3 B = ($A_1|A_2|.....|A_n$), where B is an or_parallel block.

- A_i BD B ,1 $\leq$ i $\leq$n
- $\exists$i (B CD A_i) $\wedge$ ($\forall$j((B CD A_j) $\vee$ (B CAD A_j))), j$\neq$i
- $\forall$i(B AD A_i)

Semantics 4 B = ($A_1||A_2||.....||A_n$), where B is an xor_parallel block.

- A_i BD B ,1 $\leq$ i $\leq$n
- $\exists$i (B CD A_i) $\wedge$ ($\forall$j(A_j ACD A_i)), i$\neq$j
- $\forall$i(B AD A_i)

Semantics 5 B = ($A_1, A_2, ...A_n$), where B is a contingency block.

- A_1 BD B
- A_{i+1} BAD A_i, 1 $\leq$ i<n
- B CD A_i, 1 $\leq$i$\leq$n
- B AD A_n

Semantics 6 B = (condition(C),A_1, A_2), where B is a conditional block.

- A_1 BD(C) B
- B CD(C) A_1
- B CD($\neg$C) A_2
- B AD(C) A_1

- B AD(¬C) A_2

Semantics 7 B = (condition(C);$A_1; A_2; .....; A_n$), where B is an iterative block.

- A_1 BD(C) B
- A_{i+1} BCD A_i, 1 ≤i<n
- B CD(¬C) A_n
- B AD A_i

Semantics 8 B = (list; par_type; B_1), where B is a for_each block.

- par_type = and_parallel
  - B_1 BD B, for each element of the list
  - B AD B_1, for any element of the list
  - B CD B_1, for all the elements of the list
- par_type = or_parallel
  - B_1 BD B, for each element of the list
  - B AD B_1, for all the elements of the list
  - B CD B_1, for at least one element of the list
- par_type = xor_parallel
  - B_1 BD B, for each element of the list
  - B AD B_1, for all the elements of the list
  - B CD B_1, for only one element of the list

Semantics 9 A = (A_c, AbortList(A_c)[2]), where A_c is the compensation activity of A.

- A_c BCD A
- A_c BAD AbortList(A_c)

Semantics 10 T = T_u, where T_u is the undo task of T.

[2] AbortList(A_c) contains the list of activities whose abort cause the start of A_c.

- T_u BAD T

ACTA formalism specifies the transaction semantics of a model by presenting transaction relations with predefined dependencies. However, these dependencies are expressed at the abstract level and therefore we will use the following two primitives [2, 21] to specify intertask dependencies as constraints on the occurrence and temporal order of events:

1. $e_1 \rightarrow e_2$: If e_1 occurs, then e_2 must also occur. There is no implied ordering on the occurrence of e_1 and e_2.

2. $e_1 < e_2$: If e_1 and e_2 both occur, then e_1 must precede e_2.

The ACTA dependencies we use in specifying the block semantics are expressed in terms of these two primitives as follows:

- **Commit Dependency**(t_j CD t_i):
  $(Commit_{t_j} \rightarrow Commit_{t_i}) \wedge (Commit_{t_i} < Commit_{t_j})$

- **Commit-on-Abort Dependency**(t_j CAD t_i):
  $(Abort_{t_j} \rightarrow Commit_{t_i}) \wedge (Commit_{t_i} < Abort_{t_j})$

- **Abort Dependency**(t_j AD t_i):
  $(Abort_{t_j} \rightarrow Abort_{t_i}) \wedge (Abort_{t_i} < Abort_{t_j})$

- **Abort-on-Commit Dependency**(t_j ACD t_i):
  $(Abort_{t_j} \rightarrow Commit_{t_i}) \wedge (Commit_{t_i} < Abort_{t_j})$

- **Begin Dependency**(t_j BD t_i):
  $(Start_{t_j} \rightarrow Start_{t_i}) \wedge (Start_{t_i} < Start_{t_j})$

- **Begin-on-Commit Dependency**(t_j BCD t_i):
  $(Start_{t_j} \rightarrow Commit_{t_i}) \wedge (Commit_{t_i} < Start_{t_j})$

- **Begin-on-Abort Dependency**(t_j BAD t_i):
  $(Start_{t_j} \rightarrow Abort_{t_i}) \wedge (Abort_{t_i} < Start_{t_j})$

The guards of events corresponding to these two primitive dependencies are as follows [2, 37]:

- For the constraint $e < f$, which corresponds to the dependency $D_< = \bar{e} \vee \bar{f} \vee e \bullet f$, the guards are:

  $\mathcal{G}(e) = \mathsf{TRUE}$

  $\mathcal{G}(f) = \Diamond \bar{e} \vee \Box e$

Note that $\Box e$ means that e will always hold; $\Diamond e$ means that e will eventually hold (thus $\Box e$ entails $\Diamond e$). The $\bullet$ operator denotes sequencing in which its first argument should precede the second. $\bar{e}$ is the complement of event e denoting non-occurrence of e. For the above dependency, at runtime e can occur at any point in the history whereas f can occur only if e has occurred or it is guaranteed that $\bar{e}$ will occur.

- For the constraint $f \rightarrow e$, which corresponds to the dependency $D_\rightarrow = \bar{f} \vee e$, the guards of events are:

  $\mathcal{G}(e) = \mathsf{TRUE}$

  $\mathcal{G}(f) = \Diamond e$

These guards state that e can occur at any time in the history; f can occur if e has happened or will happen.

All the above discussion shows that we can capture intertask dependencies that determine the flow of control in the workflow system into guard expressions.

4.2 Guard Generation

We use the dependencies BD, BCD, BAD to compute start guards, AD, ACD to generate abort guards and CD, CAD to compute commit guards of activities. Note that all of these dependencies are in the form of an expression which contains one subexpression with $\rightarrow$ primitive and the other with $<$ primitive with a conjunction in between them such as $(f \rightarrow e) \wedge (e < f)$. We present the construction of guards of events e and f for this dependency in the following [37]:

Table 4.1: Guards corresponding to the dependency set

dependency	e	f	$\mathcal{G}(f)$	$\mathcal{G}(e)$
A BD B	B.start	A.start	B.start	TRUE
A BCD B	B.commit	A.start	B.commit	TRUE
A BAD B	B.abort	A.start	B.abort	TRUE
A CD B	B.commit	A.commit	B.commit	TRUE
A CAD B	B.abort	A.commit	B.abort	TRUE
A AD B	B.abort	A.abort	B.abort	TRUE
A ACD B	B.commit	A.abort	B.commit	TRUE

$$\mathcal{G}(e) = \mathcal{G}(D_{\rightarrow}, e) \wedge \mathcal{G}(D_{<}, e) = TRUE \wedge TRUE = TRUE$$
$$\mathcal{G}(f) = \mathcal{G}(D_{\rightarrow}, f) \wedge \mathcal{G}(D_{<}, f) = \Diamond e \wedge (\Diamond \bar{e} \vee \Box e) = (\Diamond e \wedge \Diamond \bar{e}) \vee (\Diamond e \wedge \Box e)$$
$$= F \vee (\Diamond e \wedge \Box e) = \Box e$$

Note that after simplification, the guard of f turned out to be $\Box e$. In other words, the occurrence of event f only requires event e to have already happened. This result facilitates the computation of the guards drastically. The guards of events of the dependency set corresponding to our workflow specification language are computed as presented in Table 4.1. Note that from this result, we conclude that if we want to compute the guard related to an activity A_1, we must consider only "A_1 ACTA_Dep A_2" type dependencies, not "A_2 ACTA_Dep A_1" type dependencies. The reason is that in the latter, the guard of any event related with A_1 is already TRUE from Table 4.1. We have omitted the $\Box$ sign since we do not have any other temporal symbol and we can treat A.e as "when event e of activity A has occurred".

If we summarize, by starting from a block structured workflow specification language, we obtain a well defined set of dependencies, all in the form $(f \rightarrow e) \wedge (e < f)$. This dependency produces straightforward guards for events. This makes it possible to compute the guards directly from the process definition with a simple algorithm. The complete guard generation process is outlined in Figure 4.1.

Knowing the dependencies implied by the block structures and the guards corresponding to each of the dependencies in the dependency set given in Table

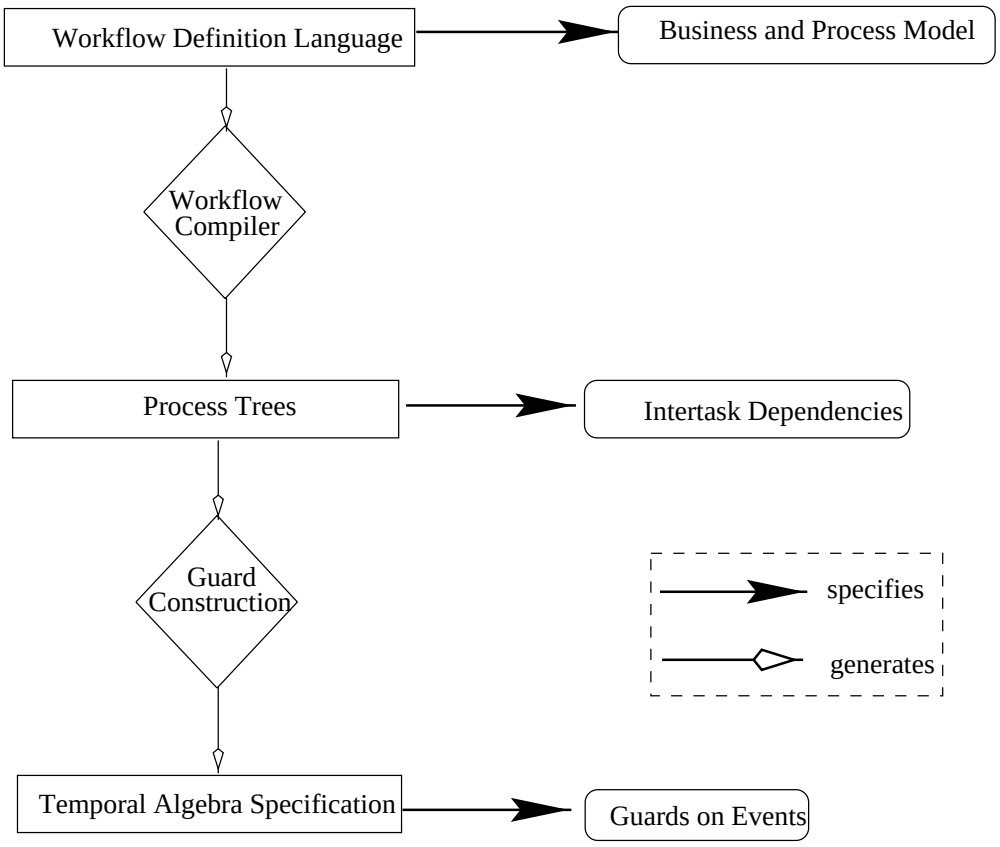


Figure 4.1: Guard generation process

4.1, it becomes easier to construct the guards of the activities by parsing the wokflow definition. First, a process tree is generated from the workflow specification. The process tree consists of nodes representing processes, blocks and tasks, and is used only during compilation time, execution being completely distributed. Each of the nodes is given a unique label to be referred in the execution phase. These activity labels make it possible for each activity instance to have its own uniquely identified event symbols. The process tree shows the dependencies between the activities of the workflow. Using Table 4.1 and the block semantics, it is possible to generate the guards of a process from its process tree. Figure 4.2 shows the process tree of the manufacturing example given in the previous chapter. Applying the guard construction algorithm on this tree given in Chapter 5, we obtain the guards listed in Table 4.2. It should be noted that in Table 4.2, some of the guards are set to TRUE right away. This is because either the occurrences

of these events do not depend on the occurrence of any other event or they are immediate(nondelayable, nonrejectable) events.

For our example process, at compile time the guards, given in Table 4.2, are generated and stored locally with the related CORBA objects that belong to each of the activities in the process, except for the activities which are labeled as 6 through 17. The reason behind this is that the number of elements in the *list* of the for_each block is unknown at compile time. Therefore, the exact number of *serial block(6)*s in for_each block is undefined at compile time. Thus, objects corresponding to the serial block node and its descendants are created at run time by generating new guards and labels.

The purpose lying behind this special block structure is to provide efficiency by executing the same actions on individual items in a list in a parallel fashion . Since the same actions will be carried out on the elements of the list and in an independent way, it is a lot more efficient to carry out these executions in parallel. But, since the number of elements in the list can not be known at compile time, the guard expressions for the activities in a for_each block can not be constructed statically. To handle this problem, we developed a mechanism in which we pre-

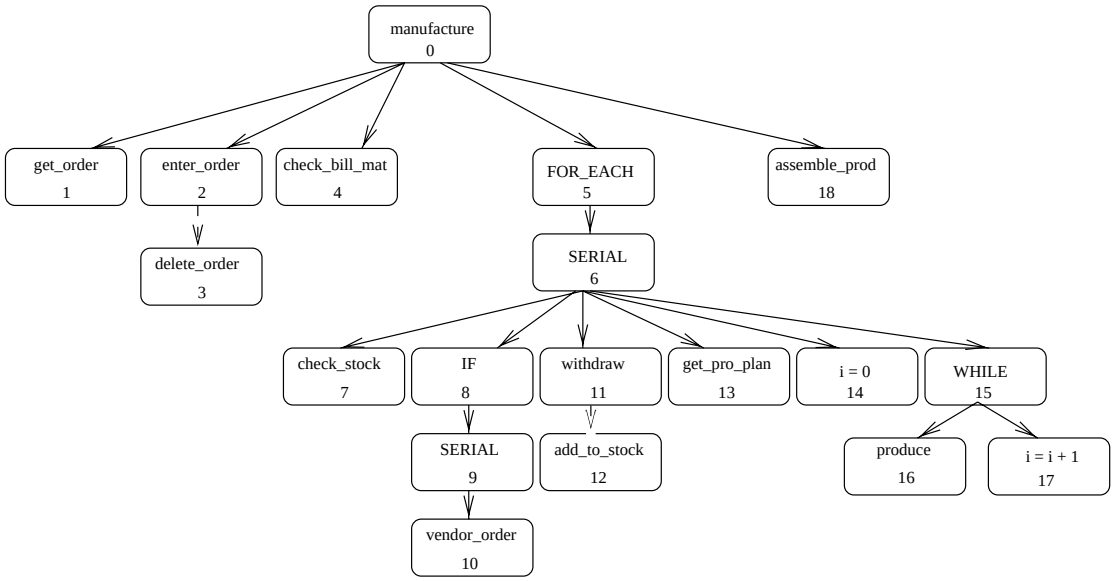


Figure 4.2: Process tree of the manufacturing example

Table 4.2: Guards of the example workflow definition

label	start guard	commit guard	abort guard
0	TRUE	18.commit	1.abort or 2.abort or 4.abort or 5.abort or 18.abort
1	0.start	TRUE	TRUE
2	1.commit	TRUE	TRUE
3	2.commit and 0.abort	TRUE	TRUE
4	2.commit	TRUE	TRUE
5	4.commit	6.commit	6.abort
6	5.start	15.commit	7.abort or 8.abort or 11.abort or 13.abort or 14.abort or 15.abort
7	6.start	TRUE	TRUE
8	7.commit	9.commit	9.abort
9	8.start and status = 0	10.commit	10.abort
10	9.start	TRUE	TRUE
11	8.commit	TRUE	TRUE
12	11.commit and 6.abort	TRUE	TRUE
13	11.commit	TRUE	TRUE
14	13.commit	TRUE	TRUE
15	14.commit	17.commit and i > no_of_steps	17.abort or 16.abort
16	15.start and i < no_of_steps	TRUE	TRUE
17	16.commit	TRUE	TRUE
18	5.commit	TRUE	TRUE

pare guard templates for the activities in the for_each block during compilation and make use of these templates at run time to generate the actual guard expressions. In order to explain the method which handles guard modifications due to for_each construct, the following simplified version of the for_each block in the manufacturing example is used:

```
FOR_EACH(part_no_list, PAR_AND)
{
    check_stock (IN part_no_list[INDEX].part_no,
                 IN part_no_list[INDEX].part_quantity,
                 OUT part_no_list[INDEX].status,
                 OUT part_no_list[INDEX].nec_quantity);
    IF (part_no_list[INDEX].status == 0) THEN
       vendor_order(IN part_no_list[INDEX].part_no,
                    IN part_no_list[INDEX].nec_quantity);
 }
```

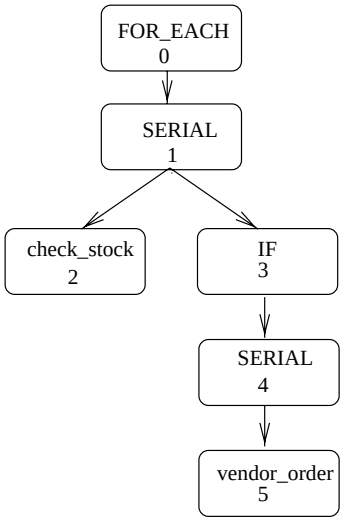


Figure 4.3: For_each process tree at compile-time

In this version, there are two tasks. *Check_stock* checks the stock database to find out if the quantity of a part in the stock is enough for production. *Vendor_order* is a user activity which is used to order a part from the vendor in case the quantity of the part in the stock is not enough. These two tasks should be applied to all of the parts in the *part_no_list* in an and_parallel fashion. That is, the body of the for_each block must be repeated for each element of the *part_no_list* in parallel. To achieve this, we consider the body of the for_each block as a serial block. During execution time, there must be as many copies of this serial block as the number of elements in the *part_no_list*. But each of these serial blocks must execute with different values of the *part_no*, *part_quantity*, *status* and *nec_quantity* variables. To provide this, the index of the part_no_list must be different for each of the serial blocks. We use the INDEX reserved word which can have different values at a time and each different value of INDEX is kept in the associated guard handlers.

The example block appears as Figure 4.3 in the process tree at compile time. And the guard templates prepared for this block during compilation are as given in Table 4.3.

At run-time, if the length of the *part_no_list* is 2, the process tree turns out to be as in Figure 4.4. In fact, the process tree is only used at compile time. The

Table 4.3: Guard templates for the for_each block

label	start guard	commit guard	abort guard
0	TRUE	1.commit	1.abort
1	0.start	3.commit	2.abort or 3.abort
2	1.start	TRUE	TRUE
3	2.commit	4.commit	4.abort
4	3.start and status = 0	5.commit	5.abort
5	4.start	TRUE	TRUE

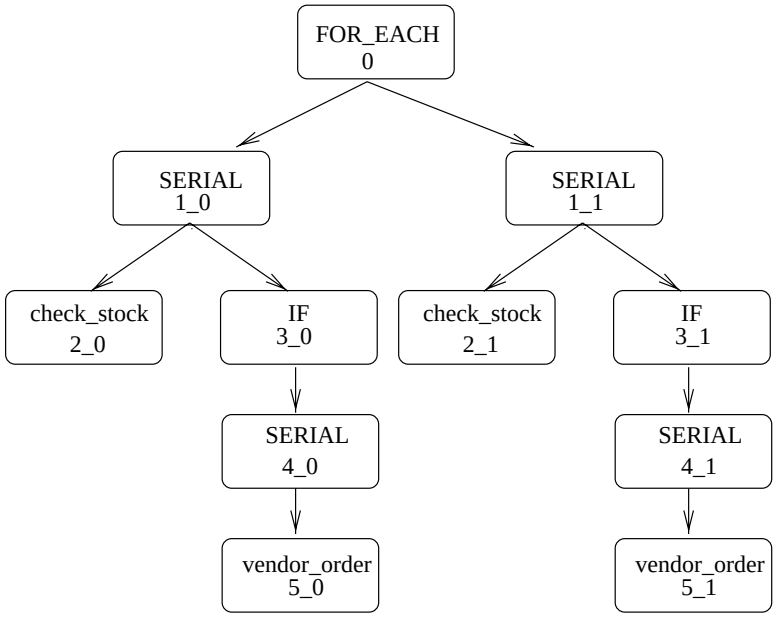


Figure 4.4: For_each process tree at run-time

Table 4.4: Guard expressions for the for_each block

label	start guard	commit guard	abort guard
0	TRUE	1_0.commit and 1_1.commit	1_0.abort or 1_1.abort
1_0	0.start	3_0.commit	2_0.abort or 3_0.abort
2_0	1_0.start	TRUE	TRUE
3_0	2_0.commit	4_0.commit	4_0.abort
4_0	3_0.start and status = 0	5_0.commit	5_0.abort
5_0	4_0.start	TRUE	TRUE
1_1	0.start	3_1.commit	2_1.abort or 3_1.abort
2_1	1_1.start	TRUE	TRUE
3_1	2_1.commit	4_1.commit	4_1.abort
4_1	3_1.start and status = 0	5_1.commit	5_1.abort
5_1	4_1.start	TRUE	TRUE

tree in Figure 4.4 is drawn to clarify how for_each block appears at run time. The guard expressions corresponding to this tree are as in Table 4.4.

As can be seen from Figure 4.4, we give new labels to each node which is under the for_each node to provide uniqueness. If we compare Table 4.3 and Table 4.4, we see that guards of the nodes that belong to the same activities in different branches of the block are similar except their labels. For example, the start guard of node labeled 2 is 1.start in Table 4.3 whereas it is 1_0.start and 1_1.start in Table 4.4. All we need to do is to change the labels of the nodes given in the guard templates according to the number of elements in the *part_no_list*. But this is not the case for the for_each node itself labeled as 0. We have to change the commit and abort guards of this node. This change can be done again according to the length of the list and the guard template but also taking the type of the parallelism into consideration.

Besides for_each block, there is another block that requires special attention. In xor_parallel block, we should guarantee that one and only one of the activities in the block body commits. To provide this, we have implemented a modified Two Phase Commitment Protocol. When xor_parallel block starts, all of its immediate children are registered to the coordinator object belonging to this block. The coordinator keeps track of status of these children to ensure that only one of them commit. In this case, the abort and commit guards need not be constructed any more for the immediate child nodes of an xor_parallel block.

4.3 Guard Handling

After the guards are constructed, an environment in which these guards are evaluated through the event occurrence messages they receive is created. Our approach associates a guard handler with each activity instance which contains the guard expressions for the significant events of that activity instance [18]. Also, there exists a task handler for each activity instance which embodies a coarse description of the activity instance including only the states and transitions (i.e. events) that are significant for coordination. A guard handler provides the message flow between the activity's task handler and the other guard handlers in the system

[20, 39]. According to the message it receives from the guard handler, a task handler causes the events related with that activity to occur. Each node in the process tree is implemented as a CORBA object with an interface for the guard handler to receive and send messages. A guard handler object consists of the following parts:

occurred event queue for start contains occurrence messages of events which affect the start guard of this guard handler.

occurred event queue for abort contains occurrence messages of events which affect the abort guard of this guard handler.

occurred event queue for commit contains occurrence messages of events which affect the commit guard of this guard handler.

start message list contains object identifiers of guard handler objects which should be informed about occurrence of the start event of the activity instance controlled by this guard handler.

abort message list contains object identifiers of guard handler objects which should be informed about occurrence of the abort event of the activity instance controlled by this guard handler.

commit message list contains object identifiers of guard handler objects which should be informed about occurrence of the commit event of the activity instance controlled by this guard handler.

start guard is a temporal expression which defines the condition under which the start event of the activity instance should occur if its start condition is satisfied.

abort guard is a temporal expression which defines the condition under which the abort event of the activity instance should occur if its abort condition is satisfied.

commit guard is a temporal expression which defines the condition under which the commit event of the activity instance should occur if its commit condition is satisfied.

start condition is a logical expression which defines the condition under which the start event of the activity instance should occur if its start guard is satisfied.

abort condition is a logical expression which defines the condition under which the abort event of the activity instance should occur if its abort guard is satisfied.

commit condition is a logical expression which defines the condition under which the commit event of the activity instance should occur if its commit guard is satisfied.

At compile time the guards are generated and stored locally with the related objects, except the activities in the for_each block. The objects to which the messages from each object are to be communicated are also recorded. A guard handler maintains the current guards for the significant events of the activity and manages communications. When a task handler is ready to make a transition, it attempts the corresponding event. Intuitively, an event can happen only when its guard evaluates to true. If the guard for the attempted event is true, it is allowed right away. If it is false, it is rejected. Otherwise, it is parked. Parking an event means disabling its occurrence until its guard simplifies to true or false. When an event happens, messages announcing its occurrence are sent to the guard handlers of other related activities. When an event announcement arrives, the receiving guard handler simplifies its guard to incorporate this information. If the guard becomes true, then the appropriate parked event is enabled [17].

4.4 Task Handling

A task handler is created for each task instance. It acts as a bridge between the task and its guard handler. The guard handler sends the information necessary

for the execution of the task, like the name of the task, parameters to the task handler and the task handler sends the information about the status of the task to the guard handler. When a task starts, its status becomes *Executing*. If it can terminate successfully, then its status is changed to *Committed* or *Done* depending on whether it is a transactional or a non-transactional task. In case the task fails, its status becomes *Abort* or *Failed*.

Task handler is a CORBA object and has a generic interface which contains the following methods to communicate with its associated guard handler:

- **Init** method is used for passing initial data such as name of the task and initial parameters to the task handler.
- **Start** method is called by the guard handler when the start guard of the task evaluates to true. This causes the task handler to invoke the actual task.

The tasks may define their status in a way that the task handler can not understand or the task may not understand the messages coming from the task handler. Therefore, it becomes essential to interfere the source code of existing tasks. If it is possible to make changes in the task, then additional calls are added to the code of the task to convert the status information and error messages so that task handler and task can understand each other. If this is not possible, then the existing task is encapsulated by a code which provides the required conversion [20].

CHAPTER 5

IMPLEMENTATION

In this chapter implementation of the guard generation algorithm is explained. First, the tools and facilities that are used for implementation are briefly introduced. Then, the program structure and the main data structures used are described. Finally, the guard generation algorithm is given.

5.1 Tools and Facilities

A workflow specification written in MFDL has been parsed using the *yacc* compiler generator, and the lexical analysis has been performed using the *lex* lexical analyzer from *Sun Release 4.1*. A CORBA compliant ORB product called *Orbix 2.1* from *IONA Technologies* [26] is used as the communication infrastructure. Orbix CORBA compiler produces C++ language mapping from the CORBA IDL programs. These IDL definitions define the interface of each of the METUFlow components. Although no CORBA object is created for the guard generation algorithm, the algorithm makes use of some of the other objects in the system by calling some of their methods.

5.2 The Program Structure

To generate the guards of the activities in a workflow specification, first we have to construct the process tree from this specification. A program called *mfdl.y*

parses the workflow specification using the *yacc* compiler generator according to the rules of the language we have designed which is given in Appendix A. During this parsing process, the process tree is also constructed. As explained in Chapter 4, process tree consists of nodes, each containing information about the activities of the workflow. Towards the end of parsing, after all the nodes are inserted into the process tree, the guard construction function is called, namely *construct_guard*. This function implements the algorithm given in Section 5.4. It is a part of the C++ program called *stree_sym.cc*. Besides implementing the guard generation algorithm, this program also contains functions for filling the arguments, parameters, message lists, conditions, nodes of the process tree. All the data type definitions are provided in a header file called *stree_sym.h*. At the end of parsing, a function called *StartExec* is called which initiates the execution of the parsed workflow process. This function uses some methods of the Guard Handler and the Task Handler objects. It first calls the *create* method of both of the components to create the actual CORBA objects. It then calls the PutInfo method of the Guard Handler so that the necessary information such as label, guard expressions, conditional expressions, message lists, arguments obtained in parsing are put into the related Guard Handler object.

5.3 The Data Structures

In this section, the definitions of the two main data structures for the nodes of the process tree and the guard expressions are given.

- **The Process Tree**

  The main process tree data structure is a structure called *TREE*. It consists of mainly the node information, several pointers to parent, compensation, undo and children nodes. Node information is kept in a structure called *NODE_INFO*. It includes identification information about the node like the label, name, task type and attributes, list of parameters, arguments, conditions of the activity that the node represents. The details of the process tree data structure are provided in the following:

```
typedef struct NODE_INFO {
        int label; // a unique label
        int type;  // activity type(block, process or task)
        int type2; // task type(transactional, user, etc)
        int type3; // task attribute(vital, critical, etc)
        char name[50];
        PARAMETER_LIST *prm;
        ARGUMENT_LIST arg;
        int retry_num; // how many retries
        Node *retry_con; // retry condition
        Node *cond; // the conditional expression
        Expr *expr; // if the node is an assignment node
        int WhileFlag; // a while block
        int ForEachFlag; // a for_each block
        int ForEachP; // the type of parallelism in
                      // for_each block
        Common::ActivityType hnt; // history node type
} NODE_INFO; // information kept in the nodes of the
             // process tree

typedef struct TREE {
        NODE_INFO info;
        TREE *parent; // pointer to parent node
        TREE *comp; // pointer to compensation
        TREE *undo; // pointer to undo
        TREE *children[50]; // pointers to
                            // immediate children
        int noofchild;
        int  flag; // normal, undo or compensation
} TREE; // the main process tree data structure
```

- **The Guard Expressions**

  The main data structure to keep the guards, namely *GuardInfo*, contains the guard expression, the conditions and message lists. Guard expression is kept in *Guard* class. This class consists of mainly a pointer to *Node* structure and several methods to evaluate the guard expression. *Node* structure represents an internal node in the guard expression tree. Nodes in this tree may be either event, variable or internal nodes. Event nodes contain status information about the significant events of the activities like *"A.start"*, denoting start event of activity A. Variable nodes contain variable or constant values. Internal nodes contain relational operators and connect any two nodes in the tree. To illustrate the guard expression tree structure,

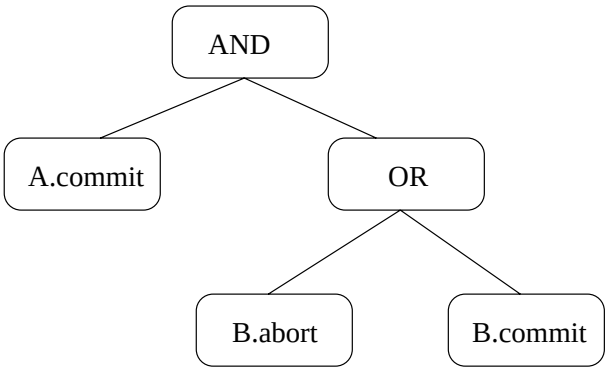


Figure 5.1: An example guard expression tree

consider the following guard expression for the start event of activity C:

G(C.start) = A.commit AND (B.abort OR B.commit)

The tree in Figure 5.1 denotes that the start of activity C requires the commitment of activity A and the termination of activity B (abort or commit).

The detailed guard expression data structure is given in the following:

```
enum GuardType {Start, Commit, Abort,Undefined};

enum ActStat {INITIAL,WAITING,NOTSTARTED,EXECUTING,COMMITTED,
              ABORTED,PREPARED,DONE,FAILED,SUSPENDED,RESUMED};
             // the execution status

enum OccVal {T,F,UNDEF}; // the occurrence value

typedef struct OccEvent {
        Common::ActStat status;
        EnactService::ActLabel Label;
        EnactService::OccVal val;
        OccEvent *back;
        OccEvent *next;
} OccEvent; // occurred event list

typedef struct EventNode {
        EnactService::ActLabel label;
        Common::ActStat status;
        EnactService::OccVal val;
        Node *parent;
} EventNode; // node about an event

typedef struct VarNode {
        int type;    // value or variable
```

```
        char *name;
        Common::ArgValue val;
        ENode *parent; // pointer to parent
} VarNode; // node about a variable

typedef struct Node {
        EnactService::RelOp op; // relational operator type
        ChildType *children; // pointer to child
                             nodes
        int NoOfChildren;
        Node *parent; // pointer to parent
        EnactService::OccVal val; // occurrence value
} Node; // internal node

typedef struct ChildType {
        int Type; // internal or event node
        union {
                 Node *n;
                 EventNode *en;
               } UType;
} ChildType;

typedef struct EChildType {
        int Type; // internal or variable node
        union {
                ENode *n;
                VarNode *vn;
              } UType;
} EChildType;

typedef struct ENode {
        EnactService::RelOp op;
        EChildType *children;
        int NoOfChildren;
        ENode *parent;
        EnactService::OccVal val;
} ENode; // event node

typedef struct EventList {
        EnactService::ActLabel label;
        struct EventList *next;
} EventList; // event list

typedef struct Expression {
        int type; // positive or negative
        ENode *root;
        EnactService::OccVal val;
} Expression; // expression about the occurrence
                of an event
```

```
class Guard {
public:
        Node *root;
        EnactService::OccVal val;
        EventNode **EventNodeList;
        int NoofEventNode;
        EnactService::ActLabel label;

        int AllDefined(Node *n);
        int AllTrue(Node *n);
        int OneTrue(Node *n);
        int OneFalse(Node *n);
        void EvaluateEventNodes(EnactService::ActLabel label,
                Common::ActStat s,EnactService::OccVal val);
        void evaluate(OccEvent oc);
        void RecurEval(Node *n); // recursive evaluation
        char *get_first_label();
}; // guard class

typedef struct GuardInfo {
        Guard g;
        Expression condition;
        EventList *EffectList;
} GuardInfo; // the main guard structure containing
             // the conditions and message lists as well
```

5.4 The Guard Generation Algorithm

In order to describe guard generation algorithm, the notations in Table 5.1 are introduced.

To each node of the process tree other than compensation and undo nodes, the following algorithm is applied:

Let t be a node in the process tree;

```
Construct_Guards(t)
begin
   Construct_StartGuard(t)
   Construct_AbortGuard(t)
   Construct_CommitGuard(t)
   if has_compensation(t) then
   begin
      Construct_Guards(t.comp)
   end
   if has_undo(t) then
```

Table 5.1: Notations

t.comp	compensation of node t
t.undo	undo of node t
has_compensation(t)	t_c is defined
has_undo(t)	t_u is defined
is_compensation(t)	t is a compensation node
is_undo(t)	t is an undo node
is_process(t)	t is a process node
is_serial(t)	t is a serial node
is_iterative(t)	t is an iterative node
is_and_parallel(t)	t is an and_parallel node
is_xor_parallel(t)	t is an xor_parallel node
is_or_parallel(t)	t is an or_parallel node
is_contingency(t)	t is a contingency node
is_conditional(t)	t is a conditional node
CreateNode(op,n)	creates an operation node with n children
CreateEventNode(t,event)	creates a node denoting the occurrence of event for node t
LeftSibling(t)	returns the child before node t
FirstChild(t)	returns the first child of node t
is_FirstChild(t)	t is the first child of its parent
LastChild(t)	returns the last child of node t
is_LastChild(t)	t is the last child of its parent
find_index(t)	returns n if t is the nth child of its parent
handle_condition(t)	constructs the condition tree for node t

```
      Construct_Guards(t.undo)
   for i = 0 to t.noofchildren do
       Construct_Guards(t.child[i])
end
```

We represent a guard as a tree whose nodes constitute a logical exression as explained in the previous section. Leaf nodes contain occurred event names and internal nodes contain logical operations such as "and", "or". Initially, values of the nodes are undefined. If a guard contains a condition, like x>y, it is represented as a separate expression tree. At execution time, both of the trees are evaluated. The algorithm presented here emphasizes the generation of guard expressions. How condition tree is constructed is out of scope of this algorithm. We use the following functions to construct the guard trees:

```
Construct_StartGuard(t)
begin
```

```
if is_compensation(t)
begin
  if !(is_iterative(t.parent.parent)) then
  begin
     if is_serial(t.parent.parent) then
     begin
        root = CreateNode(AND,3)
        root.child[0] = CreateEventNode(t.parent, COMMITTED)
        root.child[1] =  CreateEventNode(t.parent.parent, ABORTED)
        index = find_index(t.parent) + 1
        root.child[2] = CreateNode(AND,
                        t.parent.parent.noofchildren - index)
        for i = index to t.parent.parent.noofchildren do
            if has_compensation(t.parent.parent.child[i]) then
            begin
               root.child[2].child[i] = CreateNode(OR,3)
               root.child[2].child[i].child[0] = CreateEventNode
                    (t.parent.parent.child[i].comp, COMMITTED)
               root.child[2].child[i].child[1] =  CreateEventNode
                    (t.parent.parent.child[i], ABORTED)
               root.child[2].child[i].child[2] =  CreateEventNode
                    (t.parent.parent.child[i], NOTSTARTED)
            end
     end
     else
     begin
        root = CreateNode(AND,2)
        root.child[0] = CreateEventNode(t.parent, COMMITTED)
        root.child[1] =  CreateEventNode(t.parent.parent, ABORTED)
     end
  end
```

If the grandparent of a compensation node is different than an iterative block, then it can only start if its parent has committed and its grandparent has aborted. If the grandparent is a serial block, then it should check with the right siblings of its parent whether they have compensation or not. If they have, then either their compensation should commit before the compensation node in discussion starts or need not start at all.

```
  else if is_iterative(t.parent.parent) then
  begin
       root = CreateNode(OR,2)
       if is_LastChild(t.parent) then
       begin
          root.child[0] = CreateNode(OR,1)
          root.child[0].child[0] = CreateEventNode
               (FirstChild(t.parent.parent).comp, COMMITTED)
          root.child[1] = CreateNode(AND,2)
          root.child[1].child[0] = CreateEventNode(t.parent,
                                                   COMMITTED)
```

```
            root.child[1].child[1] = CreateNode(OR,1)
            root.child[1].child[1].child[0] = CreateEventNode
                                    (t.parent.parent, ABORTED)
        end
        else
        begin
          index = find_index(t.parent) + 1
          for i = index to t.parent.parent.noofchildren do
              if has_compensation(t.parent.parent.child[i]) then
              begin
                 root.child[0] = CreateNode(OR,1)
                 root.child[0].child[0] = CreateEventNode
                              (t.parent.parent.child[i].comp)
              end
          root.child[1] = CreateNode(AND,3)
          root.child[1].child[0] = CreateEventNode(t.parent,
                                                   COMMITTED)
          root.child[1].child[1] = t.parent.parent, ABORTED)
          index = find_index(t.parent) + 1
          for i = index to t.parent.parent.noofchildren do
              if has_compensation(t.parent.parent.child[i]) then
              begin
                 root.child[1].child[2] = CreateNode(OR,3)
                 root.child[1].child[2].child[0] =
                   CreateEventNode(t.parent.parent.child[i].comp,
                                   COMMITTED)
                 root.child[1].child[2].child[1] = CreateEventNode
                      (t.parent.parent.child[i], ABORTED)
                 root.child[1].child[2].child[2] = CreateEventNode
                      (t.parent.parent.child[i], NOTSTARTED)
              end // if has_compensation
         end // if is_LastChild
    end // if is_iterative
```

If the grandparent of a compensation node is an iterative block, then the logic is the same with the grandparent serial case except that the iterations done upto that point should also be taken into account.

```
end // if is_compensation

if is_undo(t) then
begin
   root = CreateNode(OR,1)
   root.child[0] = CreateEventNode(t.parent, ABORTED)
   handle_condition(t)
end
```

If the node is an undo node, then we should only check with the parent of that node. If the parent has aborted, then undo node can start. Note that the undo condition should also be satisfied for undo node to start. This is achieved by placing the conditions into guard handler constructs.

```
else // t is neither compensation nor undo
begin
  if !(is_process(t)) then
  begin
     if is_process(t.parent) OR is_serial(t.parent) OR
        is_iterative(t.parent) then
     begin
        if is_FirstChild(t) then
        begin
           if is_process(t.parent) OR is_serial(t.parent) then
           begin
              root = CreateNode(OR,1)
              root.child[0] = CreateEventNode(t.parent, EXECUTING)
           end
           else if is_iterative(t.parent) then
           begin
             root = CreateNode(OR,2)
             root.child[0] = CreateEventNode(t.parent, EXECUTING)
             root.child[1] = CreateEventNode(LastChild(t.parent),
                                             COMMITTED)
             handle_condition{t.parent)
           end
        end
        else // not first child
        begin
           root = CreateNode(OR,1)
           root.child[0] = CreateEventNode(t.LeftSibling,
                                           COMMITTED)
        end
     end
```

If parent of t is process, serial block or iterative block, then we check if t is the first child of its parent. If so, and parent is either process or serial block, the only condition for t to start is that its parent has started execution. If the parent is iterative block, then its first child can only start if it has started and the last child of it has committed from the previous iteration. The condition of the while construct should also be taken into account. If t is not the first child, then it starts after its left sibling has committed.

```
     else if is_and_parallel(t.parent) OR
             is_xor_parallel(t.parent) OR
             is_or_parallel(t.parent) then
       begin
          root = CreateNode(OR,1)
          root.child[0] = CreateEventNode(t.parent, EXECUTING)
       end
```

For the children of parallel blocks, they can start right after their parent has started.

```
     else if is_contingency(t.parent) then
       begin
          root = CreateNode(OR,1)
```

```
            if is_FirstChild(t) then
               root.child[0] = CreateEventNode(t.parent, EXECUTING)
            else
               root.child[0] = CreateEventNode(t.LeftSibling,
                                               ABORTED)
         end
```

For a child of a contingency block to start, either it should be the first child of its parent and its parent is in execution or its left sibling has aborted.

```
      else if is_conditional(t.parent) then
           begin
              root = CreateNode(OR,1)
              root.child[0] = CreateEventNode(t.parent, EXECUTING)
              handle_condition(t.parent)
           end
```

For a child of a conditional block to start, its parent should be in execution. Besides, the conditional expression of the if construct should be satisfied.

```
     end // !(is_process)
     else //if is_process(t)
        root = CreateNode(OR,0) // no start guard
   end // if neither undo nor compensation
end // start guard generation ends

Construct_AbortGuard(t)
begin
   if is_process(t) OR is_and_parallel(t) OR is_serial(t) OR
      is_iterative(t) OR is_conditional(t) then
   begin
      root = CreateNode(OR,t.noofchildren)
      for i = 0 to t.noofchildren do
          root.child[i] = CreateEventNode(t.child[i], ABORTED)
   end
```

Abort of an activity depends on the abort of any of its children if it is a process, and_parallel block, serial block, iterative block or a conditional block.

```
   else if is_contingency(t) then
        begin
           root = CreateNode(OR,1)
           root.child[0] = CreateEventNode(LastChild(t), ABORTED)
        end
```

A contingency block aborts when its last child has aborted. (Abort of its last child implies that none of its children could manage to commit.)

```
   else if is_xor_parallel(t) OR is_or_parallel(t) then
        begin
           root = CreateNode(AND,t.noofchildren)
           for i = 0 to t.noofchildren do
```

```
            root.child[i] = CreateEventNode(t.parent.child[i],
                                            ABORTED)
      end
```

Abort of an or_parallel and xor_parallel block depends on the abort of all of its children.

```
   else if is_xor_parallel(t.parent) then
        begin
           root = CreateNode(OR, t.noofchildren - 1)
           for i = 0 to t.noofchildren - 1 do
               if t.parent.child[i] != t then
                  root.child[i] = CreateEventNode(t.parent.child[i],
                                                  COMMITTED)
        end
```

If a node is a child of an xor_parallel block, then it should abort when any of its siblings has committed. (In fact this condition is handled by the modified 2PC Protocol in our system. Guard construction for this situation is symbolic.)

```
   else
     root = CreateNode(OR,0)
end // abort guard generation ends

Construct_CommitGuard(t)
begin
   if is_process(t) OR is_serial(t) OR is_iterative(t) then
   begin
      root = CreateNode(OR,1)
      root.child[0] = CreateEventNode(LastChild(t), COMMITTED)
      if is_iterative(t) then
         handle_condition(t)
   end
```

A process or a serial block or an iterative block can commit whenever its last child has committed. Note that the conditional expression in the while construct should be checked before the iterative block commits. It should be false.

```
   else if is_and_parallel(t) then
        begin
           root = CreateNode(AND,t.noofchildren)
           for i = 0 to t.noofchildren do
               root.child[i] = CreateEventNode(t.child[i],
                                               COMMITTED)
        end
```

An and_parallel block can commit when all of its children have committed.

```
   else if is_conditional(t) OR is_contingency(t) then
        begin
           if t.noofchildren == 1 AND is_conditional(t) then
           begin
              root = CreateNode(OR,2)
```

```
            root.child[0] = CreateEventNode(t.child[0],
                                            NOTSTARTED)
            root.child[1] = CreateNode(OR,t.noofchildren)
            for i = 0 to t.noofchildren do
                root.child[1].child[i] = CreateEventNode
                                         (t.child[i], COMMITTED)
        end
```

A conditional block without an else part can commit when either its child has never started or has successfully committed.

```
        else
        begin
          root = CreateNode(OR,t.noofchildren)
          for i = 0 to t.noofchildren do
              root.child[i] = CreateEventNode(t.child[i],
                                              COMMITTED)
        end
```

A contingency or a conditional block with an else part commits when any of its children has committed.

```
      end

  else if is_xor_parallel(t) OR is_or_parallel(t) then
      begin
         root = CreateNode(AND,2)
         root.child[0] = CreateNode(OR,t.noofchildren)
         for i = 0 to t.noofchildren do
             root.child[0].child[i] = CreateEventNode
                                      (t.child[i], COMMITTED)
         root.child[1] = CreateNode(AND,t.noofchildren)
         for i = 0 to t.noofchildren do
             root.child[1].child[i] = CreateNode(OR,2)
             root.child[1].child[i].child[0] =
                  CreateEventNode(t.child[i], COMMITTED)
             root.child[1].child[i].child[1] =
                  CreateEventNode(t.child[i], ABORTED)
      end
```

If t is an xor_parallel or an or_paralllel block, then at least one of its children should commit successfully and all of its children should terminate (abort or commit).

```
  else
    root = CreateNode(OR,0)
end
```

No case is defined for the construction of guards of a for_each block. For_each block requires a run-time algorithm for the generation of guards of the nodes

in it. The method we use for this purpose consists of the preparation of guard templates at compile time and construction of actual guards at run-time using these templates. We treat a for_each block as if it were a serial block when we are preparing the templates. Then afterwards at run-time, we read those templates from a persistent storage (a database) to generate the actual guards. Generation of actual guards for a for_each block is handled by the guard handler as explained in Section 4.2.

CHAPTER 6

CONCLUSION AND FUTURE WORK

In this thesis, a guard generation algorithm for a distributed workflow enactment service based on the work presented in [36, 37] is described. Main contributions of the thesis are as follows:

- Workflow enactment service is the core component of a workflow management system because it performs the scheduling of tasks in the workflow. Distributed nature of the environments in which workflow systems execute necessitates that scheduling of tasks should also be done in a distributed fashion. The enactment service described in this thesis achieves distributed scheduling through the use of simple temporal expressions called guards.

- Block-structured and procedural nature of the workflow specification language made it possible to avoid the very general set of dependencies and their related problems during distributed scheduling of process instances. Since the set of dependencies are known beforehand, the generation of guard expressions used for scheduling of the tasks is facilitated.

- Based on the block semantics expressed through ACTA Framework, a guard generation algorithm which constructs the guards of a workflow from its specification is presented. In [2], guard generation process is said to run into combinatorial explosion, because all the possible paths for a given dependency are determined. On the other hand, the complexity of our

guard generation algorithm is n, where n is the number of nodes in the process tree (number of activities in the workflow).

- A block not only clearly defines the data and control dependencies among tasks but also presents a well defined recovery semantics, i.e., when a block aborts, the tasks that are to be compensated and the order in which they are to be compensated are already provided by the block semantics. This further enables the incorporation of recovery semantics into guards during guard generation.

This thesis presents an easy and efficient method which is used to generate the guards of events. The method makes use of the compile-time dependency information hidden in the block-structured workflow process definition. However, not always can all the dependencies be initially given at compile time. Sometimes the need may arise to modify the workflow after it has started execution. Dynamic modification of dependencies requires the generation or modification of guards at run time. In fact, the for_each block construct introduced in this thesis is an example case where guards are generated dynamically at run time. But, there may be other cases for which new methods should be developed to calculate the guard expressions while the workflow is executing. Therefore, the future work includes the identification of such cases and the development of new solutions. In addition to run-time modification of guard expressions, the future work also includes the incorporation of temporal dependencies and concurrency control dependencies into guards to give time dimension into MFDL and to control concurrency requirements distributedly.

REFERENCES

[1] G. Alonso, D. Agrawal, A. Abbadi, C. Mohan, M. Kamath and R. Guenthoer, "Exotica/fmqm: A Persistency Message-based Architecture for Distributed Workflow Management", In *Proc. of the IFIP Working Conference on Information Systems Development for Decentralized Organizations (pp 1-18)* , Trondheim, Norway.

[2] P. Attie, M. Singh, A. Sheth and M.Rusinkiewicz, "Specifying and Enforcing Intertask Dependencies", In *Proc. of the 19th International Conference on Very Large Databases (VLDB'93)*, 1993.

[3] D. Barbara, S. Mehrotra and M. Rusinkiewicz, "INCAs: Managing Dynamic Workflows in Distributed Environments", In *Journal of Database Management* , Vol.7, No.1, Winter 1996.

[4] Q. Chen and U. Dayal, "A Transactional Nested Process Management System", In *Proc. of the 12th International Conference on Data Engineering (ICDE'96)*, New Orleans, February 1996.

[5] P. K. Chrysanthis and K. Ramamritham, "ACTA: A Framework for Specifying and Reasoning about Transaction Structure and Behavior", In *Proc. of the ACM-SIGMOD International Conference on Management of Data*, 1990.

[6] P. K. Chrysanthis and K. Ramamritham, "A Unifying Framework for Transactions in Competitive and Cooperative Environments", In *IEEE Bulletin on Office and Knowledge Engineering*, 1991.

[7] P. K. Chrysanthis and K. Ramamritham, "A Formalism for Extended Transaction Models", In *Proc. of the 17th International Conference on Very Large Databases (VLDB'91)*, Barcelona 1991.

[8] P. K. Chrysanthis and K. Ramamritham, "ACTA: The SAGA Continues", In *Database Transaction Models for Advanced Applications*, Morgan Kaufmann Publishers 1992, edited by A. K. Elmagarmid.

[9] "CORBA Services: Common Object Services Specification, Transaction Service: V 1.0", OMG Document, March 1995.

[10] U. Dayal, M. Hsu and R. Ladin, "Organizing Long-Running Activities with Triggers", In *Proc. of the ACM SIGMOD*, 1990.

[11] U. Dayal, M.Hsu and R. Ledin, "A Transactional Model for Long-Running Activities", In *Proc. of the 17th International Conference on Very Large Databases (VLDB'91)*, Barcelona, 1991.

[12] A. Dogac, E. Gokkoca, S. Arpinar, P. Koksal, I. Cingil, B. Arpinar, N. Tatbul, P. Karagoz, U. Halici and M. Altinel, "Design and Implementation of a Distributed Workflow Management System: METUFlow", In *Proc. of NATO-ASI on Workflow Management Systems and Interoperability*, Dogac, A., Kalinichenko, L., Ozsu, T., Sheth, A., (Edtrs.), August 1997, pp. 60-90.

[13] A. K. Elmagarmid (ed.), "Database Transaction Models for Advanced Applications", Morgan Kaufmann, 1992.

[14] "FlowMark: Programming Guide", IBM Document No. SH19-8240-01, February 1996.

[15] A. Forst, E. Kuhn and O. Bukhres, "General Purpose WorkFlow Languages", In *Distributed and Parallel Databases* , Vol.3, No.2, April 1995.

[16] H. Garcia-Molina and K. Salem, "Sagas", In *Proc. of the ACM SIGMOD*, 1987.

[17] E. Gokkoca, M. Altinel, I. Cingil, N. Tatbul, P. Koksal and A. Dogac, "Design and Implementation of a Distributed Workflow Enactment Service", In *Proc. of the International Conference on Cooperative Information Systems*, Charleston, USA, June 1997.

[18] E. Gokkoca, "Design and Implementation of a Guard Handler for a Distributed Workflow Enactment Service", *MSc. Thesis, Dept. of Computer Engineering, Middle East Technical University*, September 1997.

[19] D. Hollingsworth, "The Workflow Reference Model", *Workflow Management Coalition Specification*, TC00-1003 (Draft 1.0), 1994.

[20] P. Karagoz, "Design and Implementation of Task Handler for a Distributed Workflow Enactment Service", *MSc. Thesis, in preparation, Dept. of Computer Engineering, Middle East Technical University*, 1998.

[21] J. Klein, "Advanced Rule Driven Transaction Management", In *Proc. of the IEEE COMPCON*, 1991.

[22] N. Krishnakumar and A. Sheth, "Managing Heterogeneous Multi-System Tasks to Support Enterprise-Wide Operations", In *Distributed and Parallel Databases* , Vol.3, No.2, April 1995.

[23] C. Mohan, G. Alonso, R. Gunthor and M. Kamath, "Exotica: A Research Perspective on Workflow Management Systems", In *Data Engineering*, Vol.18, No.1, March 1995.

[24] Object Management Group, "The Common Object Request Broker: Architecture and Specification", OMG Document Number 91.12.1, December 1991.

[25] Object Management Group, "Object Transaction Service", OMG Document Number 94.8.4, August 1994.

[26] "The Orbix Programming Guide" and "Orbix Reference Guide", Iona Technologies Ltd. Release 2.0 November 1995 P1.

[27] "Orbix Reference Guide", Iona Technologies Ltd., October 1996.

[28] M.T. Ozsu, U.Dayal and P. Valduriez, "Distributed Object Management", Morgan Kaufmann, 1994.

[29] A. Reuter and F. Schwenkreis, "ConTracts - A Low-level Mechanism for Building General-Purpose Workflow Management Systems", In *Bulletin of the TC on Data Engineering*, March 1995.

[30] M. Rusinkiewicz and A. Sheth, "Specification and Execution of Transactional Workflows", In *Modern Database Systems: The Object Model, Interoperability and Beyond*, W. Kim (ed). Addison-Wesley, 1994.

[31] M. Rusinkiewicz and A. Sheth, "Transactional Workflow Management in Distributed Systems", In *Proc. of the 1st International Workshop on Advances in Databases and Information Systems (ADBIS'94)*, 1994.

[32] F. Schwenkreis, "APRICOTS - A Prototype Implementation of a ConTract System: Management of the control flow and the communication system", In *Proc. of the 12th Symposium on Reliable Distributed Systems*, 1993.

[33] M. Shan, J. Davis, W. Du and Y. Huang, "HP Workflow Research: Past, Present, and Future", In *Proc. of NATO-ASI on Workflow Management Systems and Interoperability*, Dogac, A., Kalinichenko, L., Ozsu, T., Sheth, A., (Edtrs.), August 1997, pp. 91-105.

[34] A. Sheth and M. Rusinkiewicz, "On Transactional Workflows", In *Bulletin of the TC on Data Engineering*, June 1993.

[35] A. Sheth and K. J. Kochut, "Workflow Applications to Research Agenda: Scable and Dynamic Work Coordination and Collaboration Systems", In *Advances in Workflow Management Systems and Interoperability*, Springer-Verlag, 1997.

[36] M. Singh, "Synthesizing Distributed Constrained Events from Transactional Workflow Specifications", In *Proc. of the 12th International Conference on Data Engineering (ICDE'96)*, New Orleans, February 1996.

[37] M. Singh, "Distributed Scheduling of Workflow Computations", *Technical Report, Department of Computer Science, North Carolina State University*, 1996.

[38] R. M. Soley (ed.) and C.M. Stone, "Object Management Architecture Guide", Third Edition, John Wiley & Sons, 1995.

[39] N. Tatbul, S. Arpinar, P. Karagoz, I. Cingil, E. Gokkoca, M. Altinel, P. Koksal, A. Dogac and T. Ozsu, "A Workflow Specification Language and its Scheduler", In *Proc. of the 12th International Symposium on Computer and Information Sciences*, Antalya, October 1997.

[40] H. Wachter and A. Reuter, "The ConTract Model", In *Transaction Models for Advanced Database Applications*, Chapter 7, Morgan-Kaufmann, February 1992.

[41] D. Wodtke, J. Weissenfels, G. Weikum and A. K. Dittrich, "The Mentor Project: Step Towards Enterprise-Wide Workflow Management, In *Proc. of the 12th International Conference on Data Engineering (ICDE'96)*, New Orleans, February 1996.

[42] D. Wodtke and G. Weikum, "A Formal Foundation for Distributed Workflow Execution Based on State Charts", In *Proc. of the 6th International Conference on Database Theory*, Greece, 1997.

[43] W. M. P. van der Aalst, "Petri-Net-based Workflow Management Software", In *Proc. of the NSF Workshop on Workflow and Process Automation in Information Systems: State-of-the-art and Future Directions*, 1996.

APPENDIX A

BNF OF METUFlow WORKFLOW DEFINITION LANGUAGE

In this appendix the Backus-Naur Form (BNF) representation of METUFlow workflow definition language is given. The tokens are represented with capital letters.

```
<wfdl>                  ::= <type_defs> <actvty_defs> <process_defs>

<type_defs>             ::=
                          | <type_defs> <type_def> ';'

<type_def>              ::= TYPEDEF <wfrd_type> <id_name>
                          | TYPEDEF <wfrd_type> <id_name> '[' <number> ']'

<actvty_defs>           ::=
                          | <actvty_defs> <actvty_defn> ';'

<actvty_defn>           ::= <trans_actvty>
                          | <non_trans_actvty>
                          | <trans_2PC_actvty>
                          | <user_actvty>
                          | <subprocess>

<subprocess>            ::= PROCESS <process_name> <parameters> <duration>
                            <priority> <type1>

<trans_actvty>          ::= TRANS <actvty_name> <parameters> <duration>
                            <priority> <type1>

<actvty_name>           ::= IDENTIFIER

<type1>                 ::=
```

```
                          | TYPE NON_VITAL

<non_trans_actvty>      ::= NON_TRANS <actvty_name> <parameters>
                            <duration> <priority> <type1>

<trans_2PC_actvty>      ::= TRANS_2PC <actvty_name> <parameters>
                            <duration> <priority> <type1>
                          | TRANS_2PC <actvty_name> <parameters>
                            <duration> <priority> <type2>

<type2>                 ::= TYPE CRITICAL
                          | TYPE CRITICAL NON_VITAL

<user_actvty>           ::= USER_ACTIVITY <actvty_name> <parameters>
                            <duration> <priority> <participants>

<retry>                 ::=
                          | RETRY <number> TIMES
                          | RETRY '(' IF <condition> ')' <number> TIMES

<compensation>          ::=
                          | COMPENSATED_BY <actvty_or_process_req>
                          | COMPENSATED_BY '{' <expression> '}'

<undo>                  ::=
                          | UNDO_BY <actvty_or_process_req>
                          | UNDO_BY '(' IF <condition> ')'
                            <actvty_or_process_req>

<process_defs>          ::= <process_defs> <process_defn>
                          | <process_defn>

<process_defn>          ::= DEFINE_PROCESS <process_name> <parameters>
                            <duration> <priority>
                            '{'
                            <declaration_list>
                            <expression_list>
                            '}'

<process_name>          ::= <id_name>

<parameters>            ::= '(' ')'
                          | '(' <prms> ')'

<prms>                  ::= <prms> ',' <def_parlist>
                          | <def_parlist>

<def_parlist>           ::= IN <wfrd_type> <def_id_name>
                          | OUT <wfrd_type> <def_id_name>
                          | INOUT <wfrd_type> <def_id_name>

<def_id_name>           ::=
                          | <id_name>

<name_list>             ::= <id_name>
                          | <name_list> ',' <id_name>
```

```
<init_val>              ::= <signed_number>
                          | <char>
                          | <string>
                          | <signed_float>

<var_name2>             ::=
                          | <id_name>

<var_name>              ::=
                          | <id_name>
                          | <id_name> '=' <init_val>
                          | <id_name> '[' <number> ']'

<var_list>              ::= <var_name>
                          | <var_list> ',' <var_name>

<wfrd_type>             ::= INT_TYPE
                          | FLOAT_TYPE
                          | STRING_TYPE
                          | CHAR_TYPE
                          | OBJ_TYPE
                          | struct_type
                          | user_defined

<user_defined>          ::= <id_name>

<struct_type>           ::= STRUCTURE <var_name2>
                            '{'
                            <var_dec_list>
                            '}'

<var_dec_list>          ::= <var_dec_list> <var_dec> ';'
                          | <var_dec> ';'

<var_dec>               ::= <wfrd_type> <var_list>

<duration>              ::=
                          | DURATION <duration_expr>

<duration_expr>         ::= <number> DAYS  <hour_expr>
                          | <number> HOURS  <min_expr>
                          | <number> MINUTES <sec_expr>
                          | <number> SECONDS

<hour_expr>             ::=
                          | <number> HOURS <min_expr>

<min_expr>              ::=
                          | <number> MINUTES <sec_expr>

<sec_expr>              ::=
                          | <number> SECONDS

<priority>              ::=
                          | PRIORITY  <number>
```

```
<participants>          ::= PARTICIPANT <name_list>

<declaration_list>      ::= <declaration_list> <declaration> ';'
                          | <declaration> ';'

<declaration>           ::= <actvty_dec>
                          | <process_dec>
                          | VAR <wfrd_type> <var_list>

<actvty_dec>            ::= EXT_ACTIVITY <actvty_name> <actvty_ins_name>
                            <duration> <priority> <property>

<actvty_ins_name>       ::= <id_name>

<property>              ::=
                          | TYPE NON_VITAL
                          | TYPE CRITICAL
                          | TYPE CRITICAL  NON_VITAL

<process_dec>           ::= EXT_PROCESS <process_name> <process_ins_name>
                            <duration> <priority>

<process_ins_name>      ::= <id_name>

<expression_list>       ::= <expression_list> <expression>
                          | <expression>

<expression>            ::= <actvty_or_process_req> <retry> <undo>
                            <compensation>  ';'
                          | <assignment>
                          | <block>

<actvty_or_process_req> ::= <id_name> '(' <call_prms> ')'

<call_prms>             ::=
                          | <call_parlist>

<call_parlist>          ::= <wfrd_expr>
                          | <call_parlist> ',' <wfrd_expr>

<assignment>            ::= <field_expr> '=' <wfrd_expr> ';'

<wfrd_varname>          ::= <id_name>

<wfrd_expr>             ::= <char>
                          | <string>
                          | <signed_float>
                          | <wfrd_numeric_expr>

<wfrd_numeric_expr>     ::= <wfrd_numeric_expr> <wfrd_op>
                            <wfrd_numeric_expression>
                          | <wfrd_numeric_expression>

<field_expr>            ::= <wfrd_varname>
                          | <wfrd_varname> '[' INDEX ']'
```

```
                          | <wfrd_varname> '[' <wfrd_expr> ']'
                          | <field_expr> '.' <wfrd_varname>
                          | <field_expr> '.' <wfrd_varname> '[' INDEX ']'
                          | <field_expr> '.' <wfrd_varname>'['<wfrd_expr>']'

<wfrd_numeric_expression> ::= <field_expr>
                          | '-' <wfrd_varname>
                          | <signed_number>
                          | '(' <wfrd_numeric_expr> ')'

<wfrd_op>              ::= '+'
                          | '-'
                          | '*'
                          | '/'
                          | '%'

<block>                ::= <named_block>
                          | <iterative_block>
                          | <conditional_block>
                          | <for_each_block>

<for_each_block>       ::= FOR_EACH <block_name>
                           '(' <field_expr> ',' <parallel> ')'
                           <implicit_serial_block> <compensation>

<parallel>             ::= PAR_AND
                          | PAR_OR
                          | PAR_XOR

<named_block>          ::= <block_type> <block_name>
                           '{' <expression_list> '}' <compensation>  ';'

<block_type>           ::= SERIAL
                          | PAR_AND
                          | PAR_OR
                          | PAR_XOR
                          | CONTINGENCY

<block_name>           ::=
                          | <id_name>

<iterative_block>      ::= WHILE <condition> DO
                           '{' <expression_list> '}' <compensation> ';'

<conditional_block>    ::= IF <condition> THEN <implicit_serial_block>
                           <else_part>

<implicit_serial_block> ::= '{' <expression_list> '}' <compensation>

<else_part>            ::= ';'
                          | ELSE <implicit_serial_block> ';'

<condition>            ::= <condition> <logical_op> <comparison_expr>
                          | <comparison_expr>

<comparison_expr>      ::= <comparison>
```

```
                          | NOT <comparison>

<comparison>              ::= <wfrd_expr> <comp_op> <wfrd_expr>
                          | '(' <condition> ')'

<comp_op>                 ::= '=='
                          | '!='
                          | '<='
                          | '>='
                          | '>'
                          | '<'

<logical_op>              ::= AND
                          | OR
                          | XOR

<id_name>                 ::= IDENTIFIER

<number>                  ::= NUMBER

<signed_number>           ::= NUMBER
                          | SIGNED_NUMBER

<float>                   ::= FLOAT

<signed_float>            ::= FLOAT
                          | SIGNED_FLOAT

<char>                    ::= CHAR

<string>                  ::= YSTRING
```